Softw Syst Model (2013) 12:121–144
DOI 10.1007/s10270-011-0190-3

SPECIAL SECTION PAPER

A collection operator for graph transformation

Roy Grønmo · Stein Krogdahl · Birger Møller-Pedersen

Received: 1 December 2009 / Revised: 15 November 2010 / Accepted: 16 January 2011 / Published online: 11 February 2011

Abstract Algebraic graph transformation has a well-established theory and associated tools that can be used to perform model transformations. However, the lack of a construct to match and transform collections of similar subgraphs makes graph transformation complex or even impractical to use in a number of transformation cases. This is addressed in this paper, by defining a collection operator which is powerful, yet simple to model and understand. A rule can contain multiple collection operators, each with lower and upper bound cardinalities, and the collection operators can be nested. An associated matching process dynamically builds a collection free rule that enables us to reuse the existing graph transformation apparatus. We present model transformation examples from different modeling domains to illustrate the benefit of the approach.

Keywords Graph transformation · Model transformation · Matching

1 Introduction

Graph transformations have been proposed by several authors as a means to perform model transformations [5,9]. The graphical way to define model transformations, the available tool support [12,33,37], and the well-established theory including termination and confluence analysis [22,29] make graph transformation appealing.

The graph concept is based on nodes and (usually directed) edges from which we can define models. Many model transformations can therefore be defined by a set of graph transformation rules, where each rule consists of a left hand side (LHS) graph, a right hand side (RHS) graph, and an interface (I) graph. The elements in the interface graph are to be preserved, the elements in $LHS \setminus I$ are to be deleted, and the elements in $RHS \setminus I$ are to be added.

The simple nature of graph transformation is probably a key factor to its success, since this makes it relatively easy to implement tools and to establish theory on its concepts. For the graph transformation designer, on the other hand, the lack of higher level constructs reduces the usability of graph transformation. Hence, some authors have proposed to raise the level of abstraction by introducing new and powerful graph transformation mechanisms, e.g., *the star operator* [23] and *recursion* [16].

Our experience on a number of graph transformation examples reveals an often occurring need to match collections of similar subgraphs, which cannot be solved by the star operator or recursion. The need is addressed by our *collection operator*. The collection operator allows us to express powerful model transformations using a single rule.

Our collection operator can be seen as a generalization of *set nodes* in PROGRES [33] and *multi objects* in Fujaba [12]. Those two constructs support collection matches of single nodes only. In many cases this is too restrictive and a lot of recent approaches [1,6,11,18,24,30] address this by allowing to match collections of similar subgraphs. Our collection operator aims to be concise and easy to use for the rule designer, and at the same time expressive enough for many typical model transformation scenarios.

Communicated by Jeff Gray and Richard Paige.

R. Grønmo (✉) · S. Krogdahl · B. Møller-Pedersen
Department of Informatics, University of Oslo, Oslo, Norway
e-mail: roy.gronmo@sintef.no

R. Grønmo
SINTEF ICT, Oslo, Norway

The paper is structured as follows. Section 2 provides the formal foundation of algebraic graph transformation. Section 3 presents our collection operator. Section 4 shows some examples, including a large example, where the collection operator is valuable. Section 5 justifies the need for a collection operator by showing how complicated it is to simulate a rule with collection operators by collection free rules in the AGG graph transformation tool. Section 6 extends our approach so that a rule can have nested collection operators. Section 7 covers related work and Sect. 8 concludes.

2 Graph transformation

In this section we describe a well-known formal foundation of algebraic graph transformation [20].

Definition 1 (*Graph and graph morphism*) A *graph* $G = (G_N, G_E, src, trg)$ consists of a set G_N of nodes, a set G_E of edges, two mappings $src, trg : G_E \to G_N$, assigning to each edge $e \in G_E$ a source node $src(e) \in G_N$ and target node $trg(e) \in G_N$. A *graph morphism* $f : G_1 \to G_2$ from one graph to another, with $G_i = (G_{E,i}, G_{N,i}, src_i, trg_i)$, $(i = 1, 2)$, is a pair $f = (f_E : G_{E,1} \to G_{E,2},\ f_N : G_{N,1} \to G_{N,2})$ of mappings, such that $f_N \circ src_1 = src_2 \circ f_E$ and $f_N \circ trg_1 = trg_2 \circ f_E$ (preserve source and target).

A graph morphism $f : G_1 \to G_2$ is injective if f_N and f_E are injective mappings. Only injective graph morphisms will be relevant in this paper.

Definition 2 (*Rule*) A *graph transformation rule* $p : L \xleftarrow{l} I \xrightarrow{r} R$ consists of three graphs L(LHS), I(Interface) and R(RHS) and a pair of injective graph morphisms $l : I \to L$ and $r : I \to R$.

Definition 3 (*Match*) Given a rule $p : L \xleftarrow{l} I \xrightarrow{r} R$ and a graph G. Then an occurrence of L in G, i.e., an injective graph morphism $m : L \to G$, is called *match*. The function $isMatch : L, G, (L \to G) \to Bool$ returns true if and only if $L \to G$ is a match of L in G. A match m for rule p satisfies *the dangling condition* if no node in $m(L \setminus l(I))$ is incident to an edge in $G \setminus m(L \setminus l(I))$.

Definition 4 (*Derivation Step*) Given a graph G, a graph transformation rule $p : L \xleftarrow{l} I \xrightarrow{r} R$, and a match $m : L \to G$, then there exists a *derivation step* from the graph G to the graph H if and only if the dangling condition is satisfied. H is constructed as follows:

1. Remove the image of the non-interface elements of L in G, i.e., $H' = G \setminus m(L \setminus l(I))$.
2. Add the non-interface elements of R into H, i.e., $H = H' \cup (R \setminus r(I))$.

A *negative application condition* [20] is an extension of the LHS which prevents matches from being applied in a derivation step.

Definition 5 (*Negative Application Condition (NAC)*) A *NAC* for a graph transformation rule $L \xleftarrow{l} I \xrightarrow{r} R$, is defined by a pair of injective graph morphisms: $L \xleftarrow{s} NI \xrightarrow{t} N$, where N is *the negative graph*, and NI defines the interface graph between L and N. A match $m : L \to G$ satisfies the NAC if and only if there does not exist an injective graph morphism $n : N \to G$ which preserves the NI interface mappings, i.e., for all nodes v in NI we have $n_N(t_N(v)) = m_N(s_N(v))$ and for all edges e in NI we have $n_E(t_E(v)) = m_E(s_E(e))$. A rule can have an arbitrary number of NACs, and a derivation step can only be applied if a match satisfies all the NACs of the matched rule.

In addition to the above, we adopt the theory of *typed attributed graphs* [17], where graphs are extended by assigning types to nodes and edges, and by assigning a set of named attributes to each node type. A graph morphism must now also preserve the node and edge types, and the attribute values.

In the graph transformation rules throughout this paper we only explicitly display the LHS and the RHS graphs, while the interface graph is given by shared identifiers of elements in the LHS and the RHS. Such identifiers are displayed next to their elements.

2.1 Concrete and abstract syntaxes

Typed attributed graphs are rich enough to represent most of today's modeling languages in a natural way. The choice of graph representation is based on concepts available in the metamodel of a modeling language. These graphs use a generic layout, called *abstract syntax*, where nodes are visualized as rectangles containing the type name and a list of attributes and their values. An edge is normally visualized with an arrow, where the edge type name is placed next to the arrow. *Concrete syntax* of a modeling language uses a tailored visualization with icons and rendering rules depending on the element types.

In a natural translation of UML activity models [25] to typed attributed graphs, an activity in the concrete syntax corresponds to a node of type `Activity` in the abstract syntax. A control flow in the concrete syntax corresponds to a node of type `CFlow` and two edges of types `src` and `trg` in the abstract syntax. Figure 1 shows an activity model in concrete syntax and in the corresponding abstract syntax.

To improve the usability for the model transformation designer, we specify the transformation rules based on the concrete syntax and our examples will mainly be written in the concrete syntax. The formalization is, however, defined

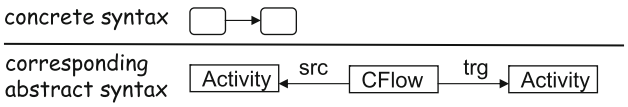


Fig. 1 An activity model in concrete syntax and corresponding abstract syntax

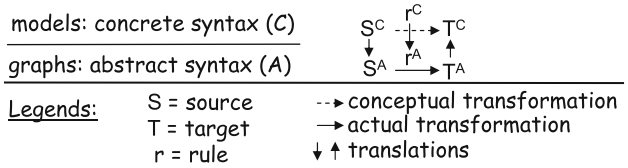


Fig. 2 Graph transformation with concrete syntax-based rules

on the abstract syntax. We assume that the translation from concrete to abstract syntax, and the translation in the opposite direction, is already defined for the relevant modeling languages. Then we can link concrete syntax-based graph transformation to abstract (and traditional) syntax-based graph transformation in a systematic way: (1) translate the concrete syntaxes of the source model and the rules (consisting of L, I, R, NI, and N models) into abstract syntax graphs, (2) apply the abstract syntax graph transformation rules on the source graph, and (3) translate the resulting abstract syntax graph back to a concrete model. Figure 2 illustrates the approach.

The approach of linking concrete syntax-based graph transformation rules to abstract syntax-based graph transformation rules has been successfully applied in our previous work [14] and by other authors [4,39] as well. However, the details of such an approach are not covered by this paper.

3 The collection operator

We propose a collection operator that can be used in a graph transformation rule to match and transform a set of similar subgraphs in one step. Figure 3c illustrates the collection operator in a workflow refactoring example [8,14]. The source model (Fig. 3a) is an activity model with two consecutive decision nodes (displayed as diamond symbols), and two inner paths leading to the activities named `doA` and `doB`. The refactored model (Fig. 3b) shows that the two decision nodes can be combined into one.

Since there can be an arbitrary number of inner paths, plain graph transformation as defined above cannot express the removal of a redundant decision node with a single rule. In Fig. 3c a single rule with the collection operator (visualized as a dotted frame) is sufficient to do the refactoring. The collection operator matches an arbitrary number of similar subgraphs, which all have an inner path leading to a single activity node between the inner decision and merge nodes.

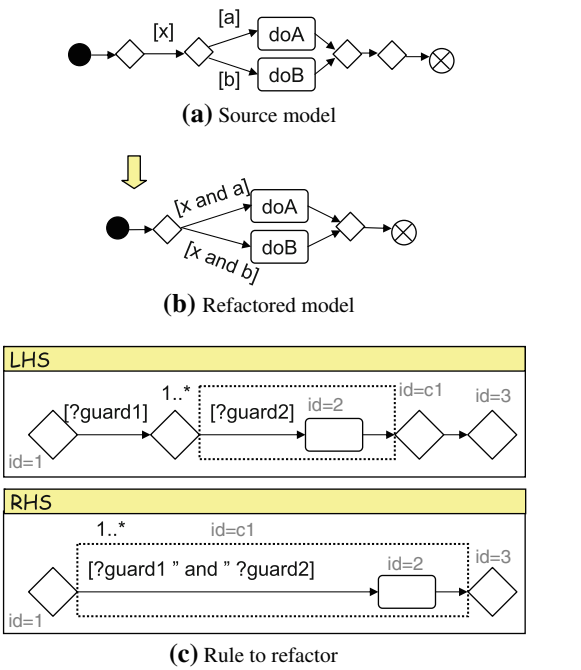


Fig. 3 Activity model refactoring: Removing redundant decision node

The outer guard (`?guard1`) is combined with each inner guard (`?guard2`) using `and` operators.

Our collection operator has a cardinality, which is indicated next to its dotted frame. The cardinality is expressed using the same notation as UML cardinalities, i.e., *lower*.. *upper*.

The size of the collection match must be greater than or equal to the lower bound cardinality (`1` in the example) in order to apply a rule. A collection match size is increased until we reach the upper bound (no limit in the example) or there are no more possible subgraph matches. The parts outside the collection operator must occur only once in a rule match.

Identifiers (e.g., `id=1`) are associated with the main elements such as activities, control nodes and control flow. Attribute variables (e.g., `?guard2`) are associated with the values of attributes such as name and guard of an activity. An identifier/variable inside a collection represents a set of identifiers/variables.

Figure 4 shows the abstract syntax representation of the concrete syntax-based rule from Fig. 3c. As previously explained, Fig. 1 shows how we map control flow and activities. A decision/merge node is mapped to a node of type `Diam` (short for Diamond). The guard is an attribute of the corresponding `CFlow` node. An id is represented as a prefix to the type. Elements that are not part of the interface do not have the id prefix.

In Fig. 5 we compare our collection operator with the PROGRES *set node* construct [33]. The four different LHS

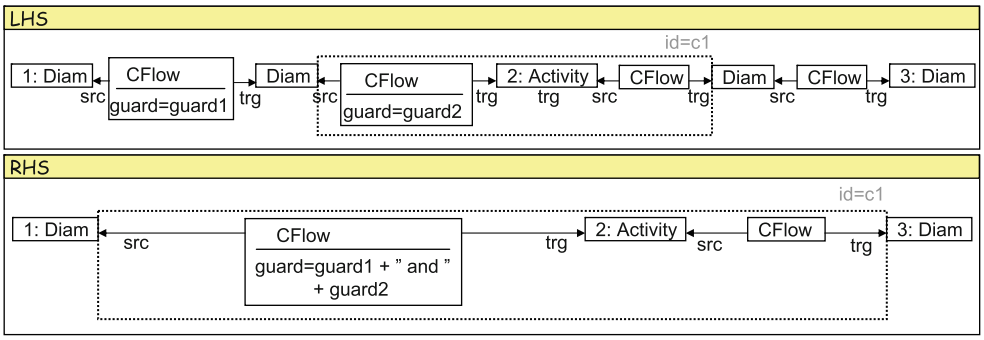


Fig. 4 Rule in abstract syntax to remove redundant decision node

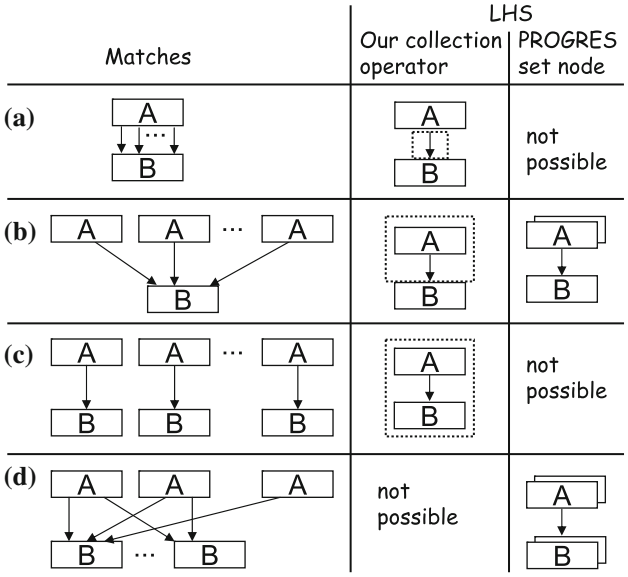


Fig. 5 Semantics of the collection operator

expressions intend to match abstract syntax with two node types A and B connected by edges, where each edge has an A typed node as its source and a B typed node as its target. As a shorthand we use the term 'A/B node' to mean 'A typed /B typed node', respectively:

case a A possible match consists of a single A node and a single B node connected by an arbitrary number of edges. With the collection operator, this is expressed by having an A node, a B node and an edge from A to B, where only the edge is inside a collection operator. This is not expressible with the set node since it only applies to nodes.

case b A possible match consists of an arbitrary number of A nodes and the same number of edges all targeting a single B node. With the collection operator, this is expressed by having an A node, a B node and an edge from A to B, where only the B node is outside of a collection operator. A set node of type A, a B node and an edge from A to B expresses the same.

case c A possible match consists of an arbitrary number of subgraphs, where each subgraph has an A node, a B node and an edge from A to B, i.e., the number of As, the number of Bs, and the number of edges are all the same. With the collection operator, this is expressed by having an A node, a B node and an edge from A to B, where all these are inside the same collection operator. This is not expressible with the set node.

case d A possible match consists of an arbitrary number of A nodes, an arbitrary number of B nodes and an arbitrary number of edges. All these three numbers can be different. With the set node, this is expressed by having an A set node, a B set node and an edge from the A set node to the B set node. This is not expressible with the collection operator.

Notice that the set node construct cannot express the rule in Fig. 4, since this is an extended version of case c.

A NAC and the RHS can only use a collection operator which is introduced in the LHS, and the correspondence is indicated by shared identifiers. The RHS indicates the changes to each collection match, and the cardinality of the collection operator must be the same in the LHS and the RHS/NACs. The actual number of collection matches of a LHS collection operator leads to the same collection instantiation number within the RHS/NACs. If the collection operator is absent in the RHS, then it implies a deletion of all the collection matches.

3.1 Multiple collection operators in the same rule

There can be multiple collection operators in the same rule. A collection operator has an identifier which is visualized next to the collection frame. No collection identifier visualization is needed in cases where the collection operator is uniquely identified by its cardinality, or when the rule has only one collection operator (e.g., Fig. 3).

Normally, a collection operator is visualized with a single frame. However, multiple frames may be used if it is impractical to use a single frame. Shared collection operator identifiers for multiple frames, e.g., in the same LHS graph, denote that these frames all belong to the same collection operator.

To avoid complexity we disallow collection operators to be overlapping. Overlapping collection operators increases the risk of producing faulty rules with unexpected results and it makes the matching and transformation more complicated. In the transformation examples we have investigated so far, we have not seen any need for such expressiveness.

There are two cases in which we define two collection operators to be overlapping. The first case, which we call *overlapping rule graph*, is when a node or an edge in the abstract syntax belongs to two different collection operators in the same rule graph ('rule graph' means the LHS, Interface or RHS graph). The second case, which we call *overlapping matches*, is when a source graph element can potentially be matched by two different collection operators. If a rule could have overlapping collection operators, then the same property of a common matched element could be updated differently by the corresponding RHS collection operators. This problem is avoided by disallowing such a rule.

Figures 6, 7, 8, 9 show four LHS activity model examples to illustrate overlapping and non-overlapping collection operators:

Figure 6 *Overlapping matches*. This LHS is not allowed since the two collection operators can have overlapping matches. In this case any individual match of collection `c1` is also an individual match of collection `c2`.

Figure 7 *Allowed LHS*. This LHS is allowed even though both collections match a single activity element. This is because the name property shall have two different values in the two collections (`doA` and `doB`).

Figure 8 *Overlapping matches*. This LHS is not allowed since the same activity can be matched by both collection operators in cases where an activity has both incoming and outgoing control flow from and to the same decision/merge node. By restricting the metamodel for UML activity models to forbid such cases, the LHS will be allowed.

Figure 9 *Overlapping rule graph*. This LHS is not allowed since the same edge is part of two collection operators in the abstract syntax, and the collection operator is formally defined in relation to the abstract syntax. Notice that the rule is not overlapping in the concrete syntax. In general, however, adjacent collection operators in the concrete syntax (e.g., this example) leads to an overlapping rule where two collection operators share at least one common edge in the abstract syntax. This fact becomes clearer in the following subsection.

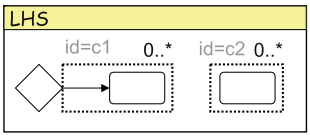


Fig. 6 *Not allowed*. An activity can be matched by both collection operators

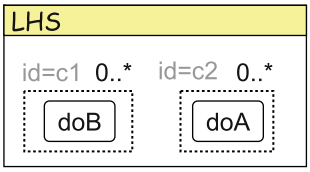


Fig. 7 *Allowed*. Possible matches are completely disjoint

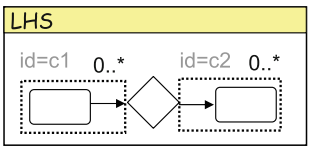
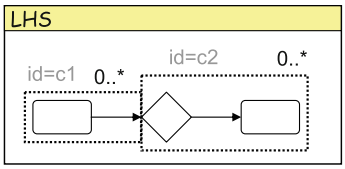
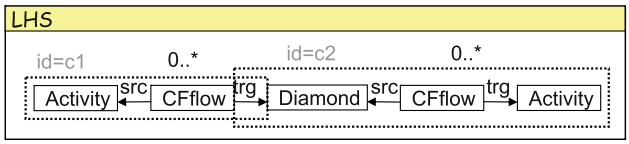


Fig. 8 *Not allowed*. An activity can be matched by both collection operators if the activity has both incoming and outgoing control flow from and to the same decision/merge



(a) concrete syntax



(b) abstract syntax

Fig. 9 *Not allowed*. Adjacent collection operators in the concrete syntax become overlapping in the abstract syntax

3.2 Mapping a collection operator from concrete to abstract syntax

In the translation from concrete syntax rules to abstract syntax rules, we must determine which abstract syntax elements belong to the collection. This is illustrated in Fig. 10 with a collection operator in concrete syntax to the left, and corresponding abstract syntax to the right.

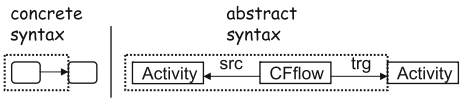


Fig. 10 A collection operator in a concrete syntax rule and its corresponding placement in the abstract syntax version of the rule

The collection operator is translated from concrete to abstract syntax according to the following three criteria:

criterion 1 If an element is inside a collection operator in the concrete syntax, then the corresponding node goes inside the collection in the abstract syntax (e.g., the `CFlow` node and the leftmost `Activity` in Fig. 10).

criterion 2 An edge connecting two nodes that are both inside a collection, belongs to the collection (e.g., the `src` edge in Fig. 10).

criterion 3 An edge in the abstract syntax connecting a collection node to a non-collection node must also be included in the collection (e.g., `trg` edge).

criterion 3 is needed because all edges shall have exactly one source and one target node (notice: source and target nodes should not be confused with the example edges of type `src` and `trg`). Otherwise, with a non-collection edge incident to a collection node, the only possible collection cardinality is 1..1, which implies that the collection is redundant.

3.3 Collection operator formalized

As we have seen above, we visualize collection operators with dotted frames surrounding the contained elements for the sake of user comprehensibility. However, in the pure graph formalism a collection operator can be represented as a node of type `coll`, with `min` and `max` as cardinality attributes. Then a set of edges, with the collection node as source, can target all the contained nodes. Incident edges of these contained nodes will implicitly also be part of the collection, as discussed in the previous subsection. Alternatively, an edge can have an attribute `inColl` with the collection node identifier as its value to explicitly denote that an edge is part of a collection.

The set of all collection operators in a rule $p : L \xleftarrow{l} I \xrightarrow{r} R$ is referred to as $Coll_p$, where $Coll_p$ is the union of the collection operators within the graphs L,I and R, i.e., $Coll_p = Coll_L \cup Coll_I \cup Coll_R$.

We use ψ to denote a function, called *cardinality mapper*, that maps each collection operator in a rule p, to a number within its cardinality range, i.e., $\psi : Coll_p \rightarrow (\mathbf{N} = \{0, 1, 2, \ldots\})$, where $\forall c \in Coll_p : \psi(c) \in [c.min, c.max]$.

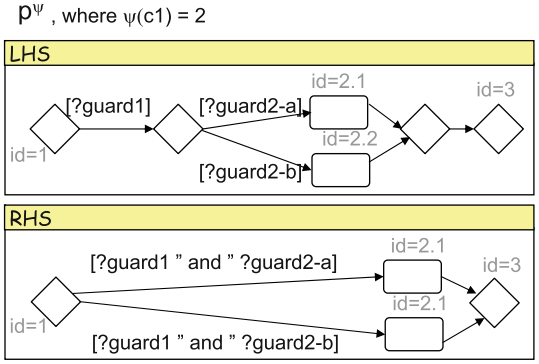


Fig. 11 Collection free rule for the rule from Fig. 3 with 2 as the ψ mapped collection size

Requirement 1 (Equal ψ mapping of interface collection operators) *We require that ψ maps similar collection operators to the same number (where 'similar' is defined by the interface). Formally,*

$$\forall c_i \in Coll_I : \psi(c_i) = \psi(l(c_i)) = \psi(r(c_i))$$

The next requirement formalizes that the RHS of a rule can only have collection operators that have been defined already in the LHS.

Requirement 2 (The RHS cannot introduce collection operators) *for a given rule $L \xleftarrow{l} I \xrightarrow{r} R$:*

$$\forall c_r \in Coll_R : \exists c_i \in Coll_I, c_l \in Coll_L : r(c_i) = c_r \ \wedge \ l(c_i) = c_l$$

Similar requirements to Requirement 2 can be provided for each NAC, since all of these can only have collection operators that have already been introduced by the LHS. Together, the two requirements above (Requirements 1 and 2) imply that ψ is defined for all collection operators in a rule if it is defined for all the LHS collection operators. Therefore it is sufficient to use $Coll_L$ in several of the formulas instead of $Coll_p$.

For a rule $p : L \xleftarrow{l} I \xrightarrow{r} R$ with at least one collection operator, we let $p^\psi : L^\psi \xleftarrow{l} I^\psi \xrightarrow{r} R^\psi$ denote the collection free rule where each collection operator c in p is replaced by $\psi(c)$ number of collection content copies. In these copies all the copied elements/attributes get fresh identifiers/variables respectively, while the interface elements between the LHS and the RHS are maintained. Similarly, $L^\psi \xleftarrow{s} NI^\psi \xrightarrow{t} N^\psi$ denotes a collection free NAC.

In Fig. 11 we have made a collection free rule p^ψ for the rule from Fig. 3c. Collection $c1$ has been mapped to 2, i.e., $\psi(c1) = 2$.

Definition 6 defines when a cardinality mapper extends another one.

Definition 6 (*Extended cardinality mapper*) Given a rule $p : L \overset{l}{\leftarrow} I \overset{r}{\rightarrow} R$ with at least one collection operator and a graph G. A cardinality mapper ψ^+ *extends* the cardinality mapper ψ (denoted $\psi^+ \succ_L \psi$) if and only if there is at least one greater collection cardinality and none of the collection cardinalities are smaller:

$$\psi^+ \succ_L \psi \overset{\text{def}}{=} \exists c \in Coll_L : \psi^+(c) > \psi(c) \\ \wedge \forall c \in Coll_L : \psi^+(c) \geq \psi(c)$$

In order to define a rule extension, we need to define that a graph is a subgraph of another graph.

Definition 7 (*Subgraph*) A graph $B = (B_N, B_E, src_B, trg_B)$ is a *subgraph* of a graph $A = (A_N, A_E, src_A, trg_A)$ if A contains all the nodes and edges of B and if A preserves all the src and trg mappings from B. Formally,

$$A \supseteq B \overset{\text{def}}{=} A_N \supseteq B_N \wedge A_E \supseteq B_E \\ \wedge \forall e \in B_E : src_A(e) = src_B(e) \wedge trg_A(e) \\ = trg_B(e)$$

A rule extends another rule if the graphs of the second rule are all subgraphs of the corresponding graphs of the first rule, and the first rule contains additional elements due to an extended cardinality mapper.

Definition 8 (*Extended rule*) A rule p^{ψ^+} *extends* the rule p^{ψ} (denoted $p^{\psi^+} \supset p^{\psi}$) if and only if ψ^+ extends the cardinality mapper ψ and if the L, I and R graphs of p^{ψ} are subgraphs of the corresponding graphs of p^{ψ^+}. Formally,

$$p^{\psi^+} \supset p^{\psi} \overset{\text{def}}{=} \psi^+ \succ_L \psi \wedge L^{\psi^+} \\ \supseteq L^{\psi} \wedge I^{\psi^+} \supseteq I^{\psi} \wedge R^{\psi^+} \supseteq R^{\psi}$$

A morphism extends another morphism if the first morphism is based on an extended rule of the other one and have similar morphism for the LHS subgraph.

Definition 9 (*Extended morphism*) An injective morphism $m^{\psi^+} : L^{\psi^+} \rightarrow G$ *extends* the injective morphism $m^{\psi} : L^{\psi} \rightarrow G$ (denoted $m^{\psi^+} \supset m^{\psi}$) if and only if p^{ψ^+} extends the rule p^{ψ} and if the morphisms of m^{ψ} are preserved by m^{ψ^+}. Formally,

$$m^{\psi^+} \supset m^{\psi} \overset{\text{def}}{=} p^{\psi^+} \supset p^{\psi} \wedge m^{\psi^+}(L^{\psi}) = m^{\psi}(L^{\psi})$$

Definition 10 defines a match for rules with collection operators, where such a match must be non-extendable.

Definition 10 (*Match for a rule with collection operators* (*cMatch*)) Given a rule $p : L \overset{l}{\leftarrow} I \overset{r}{\rightarrow} R$ with at least one collection operator, a graph G, and a cardinality mapper ψ. An injective morphism $m^{\psi} : L^{\psi} \rightarrow G$ is a *cMatch* of rule p in G if and only if m^{ψ} is a non-extendable injective morphism in G. Formally,

$$isCMatch(L, G, \psi, m^{\psi}, L^{\psi}) \\ \overset{\text{def}}{=} isMatch(L^{\psi}, G, m^{\psi}) \wedge \nexists m^{\psi^+} \in (L^{\psi^+} \rightarrow G) : \\ (m^{\psi^+} \supset m^{\psi}) \wedge isMatch(L^{\psi^+}, G, m^{\psi^+})$$

When we have a cMatch $m^{\psi} : L^{\psi} \rightarrow G$ for a rule p with collections, then m^{ψ} is also a match in the collection free rule $p^{\psi} : L^{\psi} \overset{l}{\leftarrow} I^{\psi} \overset{r}{\rightarrow} R^{\psi}$ where Definition 4 for derivation steps is still valid. We also get collection free NAC definitions as $L^{\psi} \overset{s}{\leftarrow} NI^{\psi} \overset{t}{\rightarrow} N^{\psi}$, where Definition 5 applies.

We use the notation $X \restriction \overline{c}$ to denote the set of all nodes and edges in the non-collection part of a graph X, where X is either the L,I or R graph in a rule $L \overset{l}{\leftarrow} I \overset{r}{\rightarrow} R$. For a given collection operator c, we use the notation $X \restriction c$ to denote the set of all nodes and edges that are contained in collection operator c within rule graph X.

The next two requirements formalize what it means for a rule to have non-overlapping collection operators, and cover the two cases that were previously described informally in Sect. 3.1.

Requirement 3 (Non-overlapping rule graphs) *An element belongs to at most one collection operator for a rule* $L \overset{l}{\leftarrow} I \overset{r}{\rightarrow} R$. *Formally,*

$$\forall c_1, c_2 \in Coll_X, \forall e \in (X_N \cup X_E) : \\ c_1 \neq c_2 \Rightarrow \neg(e \in c_1 \wedge e \in c_2)$$

where $X \in \{L, I, R\}$.

Requirement 4 (Non-overlapping matches) *The possible matches of two collection operators in a rule,* $L \overset{l}{\leftarrow} I \overset{r}{\rightarrow} R$, *must be non-overlapping. Formally,*

$$isMatch(L^{\psi}, G, m_1) \wedge isMatch(L^{\psi}, G, m_2) \\ \wedge \forall e \in L \restriction \overline{c} : m_1(e) = m_2(e), \forall c_1, c_2 \in Coll_L : c_1 \neq c_2 \\ \Downarrow \\ m_1(L^{\psi} \restriction c_1) \cap m_2(L^{\psi} \restriction c_2) = \emptyset$$

for arbitrary L^{ψ} *and arbitrary source graph* G, *and two injective morphisms* $m_i : L^{\psi} \rightarrow G$, *where* $i \in \{1, 2\}$. $m(S)$ *produces the set* $\{m(s) | s \in S\}$.

The next requirement ensures that interface elements must stay within the non-collection part or within the 'same' collection operator in all the three rule graphs.

Requirement 5 (Fixed collection operator) *For an interface element in a rule* $L \overset{l}{\leftarrow} I \overset{r}{\rightarrow} R$, *one of the following two alternatives must hold:*

1. *the interface element is in the non-collection part of the L,I and R graphs* (*expressed by the first disjunction in the formal definition below*), *or*
2. *the interface element is part of the same collection within the L,I and R graphs* (*expressed by the second disjunction in the formal definition below*)

Formally,

$$\forall e \in (I_N \cup I_E) \; : \; (l(e) \in L \restriction \overline{c} \; \wedge \; e \in I \restriction \overline{c} \; \wedge \; r(e) \in R \restriction \overline{c}) \\ \vee \exists c \in Coll_I : l(e) \in L \restriction l(c) \\ \wedge \; e \in I \restriction c \; \wedge \; r(e) \in R \restriction r(c)$$

Similarly, the interface elements between each NAC and the LHS must either belong to no collection operator, or it must belong to a fixed collection operator.

3.4 The matching process for rules with collection operators

This section describes an algorithm that can lay the foundation for a tool supported implementation of rules with collection operators. The minimal configuration of ψ for which we can find a cMatch for a rule $p : L \leftarrow I \rightarrow R$ with collection operators, is when $\forall c \in Coll_L : \psi(c) = c.min$. We refer to this minimal configuration of ψ as ψ^-. For a graph G, the following *sequentially ordered steps* can be used to find a cMatch in p and try to apply a derivation step for that cMatch:

1. Look for an injective morphism $m^{\psi^-} : L^{\psi^-} \rightarrow G$ in the collection free rule p^{ψ^-}.
2. Extend (if possible) the injective morphism m^{ψ^-} until it is a non-extendable injective morphism $m^{\psi} : L^{\psi} \rightarrow G$, i.e., a cMatch for p. The extension process can be achieved by iterating over each collection operator $c \in Coll_L$ and increasing $\psi(c)$ as much as possible. $\psi(c)$ can only be increased by 1, if the injective morphism can be extended with an additional subgraph match of the collection content in c.
3. Apply a derivation step with the collection free rule p^{ψ} and the match m^{ψ} if m^{ψ} satisfies all the NACs and the dangling condition. Notice that the NACs are checked only after we have produced a non-extendable match in the previous step.

The process described dynamically builds a collection free rule p^{ψ}. There can be infinitely many p^{ψ} rules since there may be collection operators without upper bound. This is why the rule p^{ψ} is dynamically built as part of the matching process.

We use a transformation task of state machine refactoring [35] to illustrate the proposed matching process above. The refactoring applies to cases where all the inner states of a composite state have outgoing transitions to the same state, and all these outgoing transitions share the same trigger and effect, while the guards must all be undefined or equivalent. In such cases we can replace all these outgoing transitions by a single transition from the composite state to the external state.

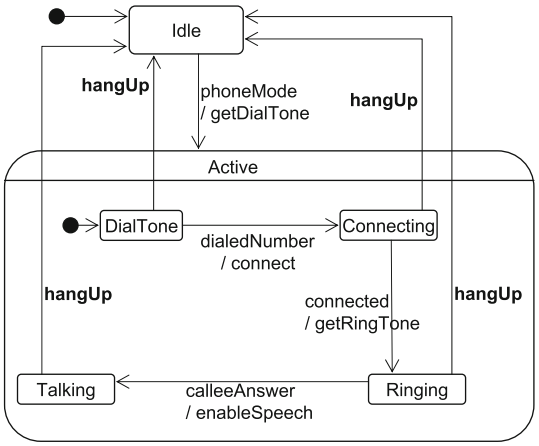


Fig. 12 State machine model of a smartphone

Figure 12 shows an example state machine that models the behavior of a smartphone (based on [5]). The state called `Idle` represents a waiting state of a smartphone. The signal `phoneMode` triggers a composite state named `Active` in which we can make phone calls. All the inner states have a trigger with the same outgoing trigger `hangUp` targeting the outer `Idle` state.

Figure 13 shows a transformation rule, named p, that defines the refactoring. The rule uses a collection operator with id `c1`, where a transition and its three attributes are inside and outside the collection, respectively. Recall that the parts outside the collections occur once, and therefore, they must have the same value for all the transitions. When the variables or values must be shared by collection nodes, we call them *shared variables* (e.g., `?trigger` and `?effect`) or *shared values* (e.g., `#null`). We have introduced a keyword `#null` to indicate that all the guard values shall be undefined (the same interpretation as a `true` value).

We have included a NAC to ensure that all the substates within the composite state have the requested transitions to the external state. The matching is injective, which means that the NAC prohibits the existence of other substates than those already matched by the LHS and repeated with `id=1` in the NAC. The new transition in the RHS gets the same trigger and effect values as those shared by all the replaced transitions, and it gets an undefined guard value.

Figure 14 illustrates how the matching algorithm works. First we non-deterministically find a match m^{ψ^-} (shown in Fig. 14a) of the rule m^{ψ^-}, which is an injective morphism for the rule p. The injective morphism m^{ψ^-} is extended by three

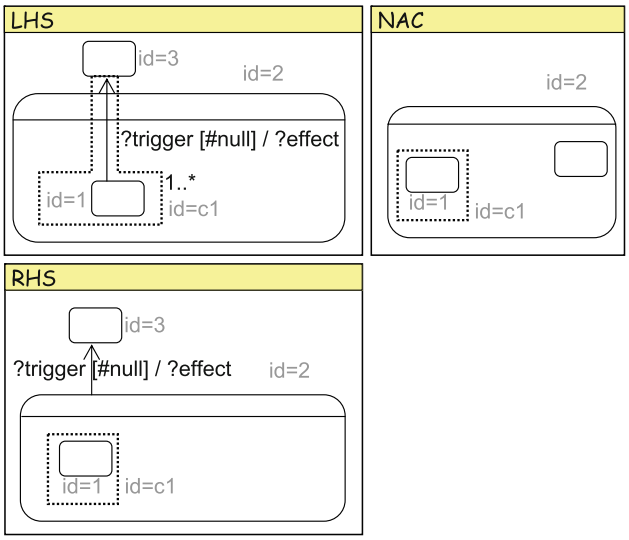
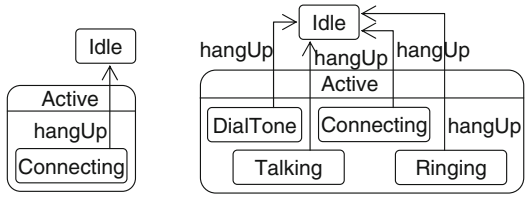


Fig. 13 Rule to refactor a state machine model by folding outgoing transitions



(a) An injective morphism m^{ψ^-}, where $\psi^-(c1) = 1$

(b) m^{ψ^-} is extended three times until we reach a cMatch, where $\psi(c1) = 4$

Fig. 14 Building a non-extendable match for the refactoring rule

subgraph matches of the collection content until we reach the cMatch m^{ψ} (shown in Fig. 14b). Figure 15 shows the refactored model after applying the match m^{ψ} and the rule p^{ψ} on the source model from Fig. 12.

4 Examples

In this section we show some examples where the collection operator is helpful. First we give three simple examples consisting of one rule each, and finally we present a larger example consisting of five rules.

4.1 Firing transitions in Petri nets

A Petri net model consists of *places*, *transitions* and directed arcs. A directed arc goes from a place to a transition or from a transition to a place. A transition T_1 has a *preset* of places consisting of each place that has a directed edge to T_1, and T_1 has a *postset* of places consisting of each place that has a directed edge from T_1. At any moment a number of *tokens* are

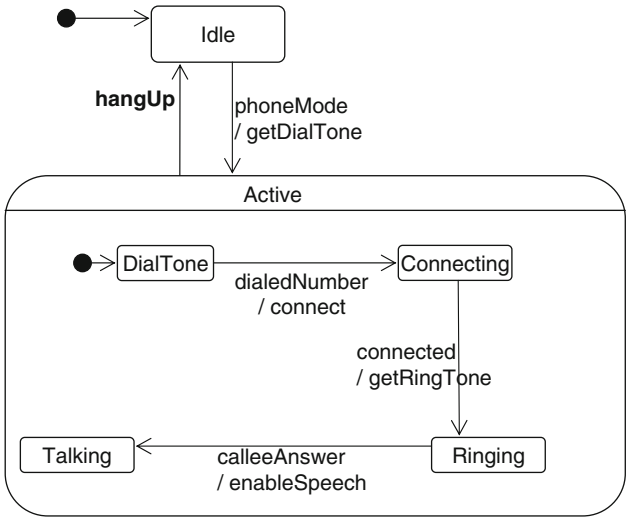


Fig. 15 Refactored state machine model

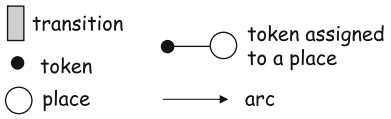
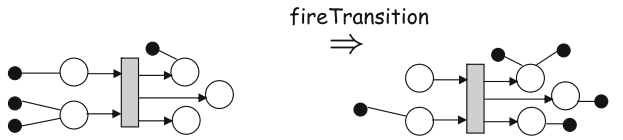


Fig. 16 Our concrete syntax for Petri nets



(a) An example Petri net

(b) The Petri net after transition firing

Fig. 17 The effects of firing a transition on a Petri net model

assigned to each place, and each token is assigned to exactly one place.

In our concrete syntax (Fig. 16), the tokens are depicted as small, filled circles, places are drawn as larger, unfilled circles, and transitions are drawn as rectangles. An example is shown in Fig. 17a, where we have a single transition consisting of two places in the preset and three places in the postset. The places in the preset have one and two tokens, respectively. The places in the postset have one, zero, and zero tokens, respectively.

A transition is *enabled* when all the places in the preset of a transition have at least one token. The transition, within the model in Fig. 17a, is thus enabled and we can *fire* a transition. When firing a transition we shall remove one token from each place in the preset and add one token to each place in the postset. The resulting model after firing the transition is shown in Fig. 17b.

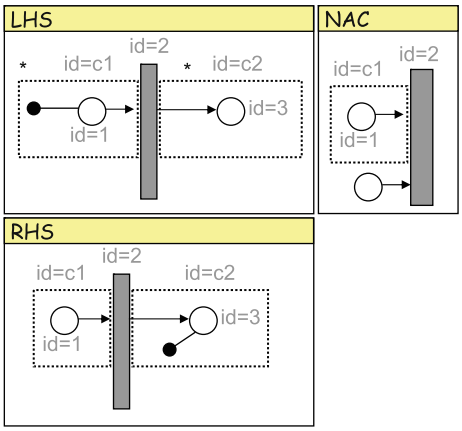


Fig. 18 A rule to fire a transition in a Petri net

With two collection operators (identified as $c1$ and $c2$) we can define the firing of a transition by a single rule (Fig. 18). Collection $c1$ expresses that we remove one token from each place in the preset, while collection $c2$ expresses that we add one token to each place in the postset. The NAC ensures that there are no preset places without a token.

For the rule to work properly and to be applicable without overlapping matches (see Requirement 4), we must require that no place can be in both the preset and postset of the same transition.

4.2 Activity model refactoring: add fork

UML activity models allow an activity to have multiple outgoing control flows, which are interpreted as an implicit fork. It is normally encouraged to use an explicit fork node instead, which we can introduce by the rule in Fig. 19.

We assume that the rule editor is more flexible than typical activity model editors, by allowing a control flow without a target. The missing target allows any kind of target node type in the model match. If a target node is required in the editor, then we can use an abstract supertype from the UML metamodel representing the possible target nodes. This type will be displayed by the abstract syntax as the target node in the rule. The lower cardinality of the collection operator is 2, so that the fork node is only introduced when there is at least two outgoing control flows.

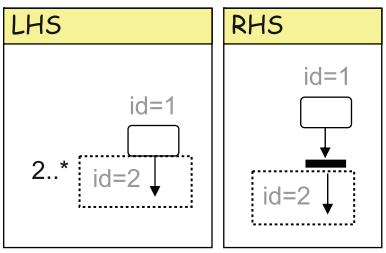


Fig. 19 Activity model refactoring: Add fork

4.3 From feature models to BPMN

This example consists of several rules that we have defined to transform feature models [2] to Business Process Modeling Notation (BPMN) [26] (BPMN models are very close to the previously described UML 2 activity models).

Here, we show only one of the rules, where we needed two collection operators (Fig. 20). The rule needs two collection operators. It is assumed that the sibling features of the feature model represent independent tasks. The rule is simplified compared to the complete rule that works recursively when the child features themselves also are parent features.

The LHS contains the to be matched feature model extract. A feature is depicted by a rectangle. A mandatory child feature is depicted by a filled circle, while an optional child feature is depicted by a non-filled circle.

The RHS contains the to be produced BPMN model extract. A start symbol is depicted by a large circle with thin line, and an end symbol is depicted by a large circle with thick line. Control flow is depicted by arrows.

Features are mapped to BPMN activities. Activities of child features are placed inside independent control flow branches of an activity. We use two collection operators, one for optional tasks and the other for mandatory tasks. Fork and join are depicted with a diamond symbol with a plus sign inside. Decision and merge are depicted by a diamond symbol with a circle inside.

In our feature metamodel a feature cannot be both an optional and a mandatory child of the same parent feature. Hence, possible matches of the two collection operators are guaranteed to be non-overlapping according to Requirement 4, which means that the specified rule is allowed.

A parent feature with the variable `?F` to match an arbitrary name is mapped to an activity node with the same name. We get an internal `fork-join` branch to represent all the mandatory tasks, and an internal `inclusive decision-merge` branch to represent all the optional tasks.

4.4 Remove unstructured cycles

This example is a business process model of a Web-based shopping application taken from Koehler et al. [19] and

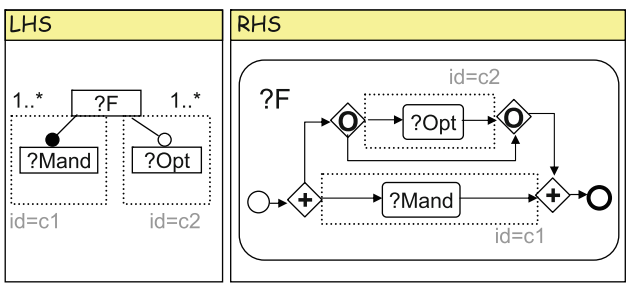
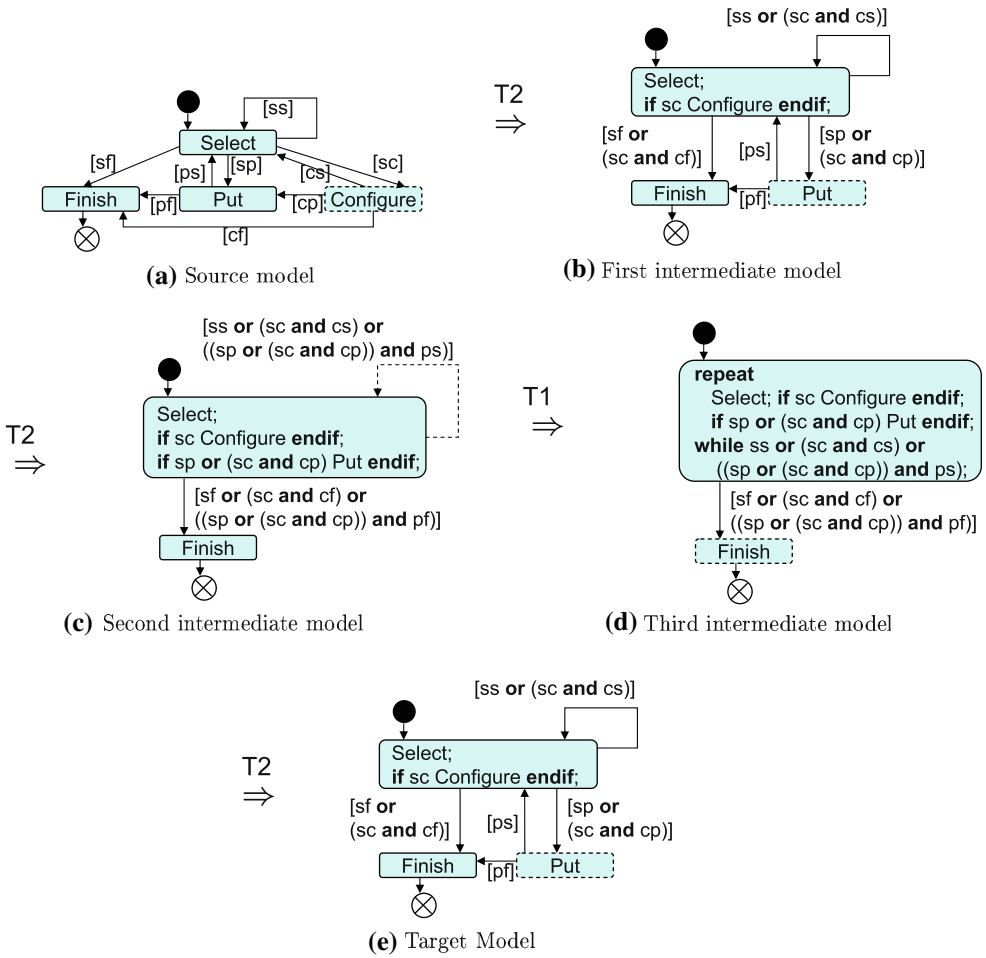


Fig. 20 From feature models to BPMN



(a) Source model

(b) First intermediate model

(c) Second intermediate model

(d) Third intermediate model

(e) Target Model

Fig. 21 From unstructured cycles to structured loops

reports a case study from IBM Zurich. The example is modeled with UML 2 activity models and we only show a submodel of the full business process model. Figure 21a shows our source model with four activities. A Web shopper is allowed to select items (`Select` activity), configure the chosen items (`Configure` activity), put chosen items into the shopping cart (`Put` activity), and to finalize the shopping by leaving with an empty cart or with items to buy (`Finish` activity). Each control flow between two activities has a two letter guard that reflects the user choice to move from one activity to the next. The two letters are the first letters in the involved activity names.

The transformation task, explained in the following, requires that there is no explicit parallelism (no forks), and no implicit parallelism resulting from multiple outgoing control flow edges from the same activity. With no parallelism we can simplify the model, as we have in the figure, by not using explicit decision nodes and interpret multiple outgoing control flow as XOR-behavior. This interpretation is different from the activity model semantics, but is unproblematic since a complete transformation would isolate this simplification to the intermediate models. All the three transformation languages benefit from the simplification.

There are several approaches where the business process models are used to automatically generate BPEL code [28,34] that can be used to execute the business process in a BPEL engine. However, BPEL does not support unstructured cycles, meaning that we need to remove all the unstructured cycles of the activity model before generating BPEL.

In an *unstructured cycle* there is more than one entry or exit point into or out of the cycle. For instance, the model in Fig. 21a contains the cycle `Select-Configure-Put-Select` which can be exited to the `Finish` activity from all three activities in the cycle. Another example is the cycle `Select-Put-Select` which can be entered from both the initial node and the `Configure` activity, and exited from both activities in the cycle.

Forcing the business process designer not to use unstructured cycles, but structured loops instead, is a heavy burden to put on the business process designer. Avoiding unstructured cycles, as in our starting model, is often a non-trivial and complex task.

Fortunately, an automatic transformation of graphs, like the shopping model (Fig. 21a), into a graph with structured loops (and no unstructured cycles), is well-known from compiler theory. Two tasks, called T1 and T2, can be applied non-deterministically until neither is applicable.

The tasks T1 and T2 will reduce the number of activities and control flow, while expanding the activity nodes from plain activities to become structured activities. We will use the name property of the activities to represent structured activities with arbitrarily many `repeat-while` and `if` expressions. At the end we have a single structured activity with no explicit control flow, only hidden control flow in the name attribute value (the model in Fig. 21e). It is straightforward to translate from the hidden control flow of `repeat-while` and `if` expressions in the name attribute of an activity, into plain activities with explicit control flow, by introducing decision and merge nodes. The end result is then guaranteed to be without unstructured cycles.

In order to apply a transformation based on the tasks T1 and T2, the source model must have the following three characteristics: (1) the model contains at least one unstructured cycle, (2) there is no parallelism, and (3) the model represents a *two-terminal region*. A source model is called a *two-terminal region* if it has a single initial node and a single final node. Our source model from Fig. 21a satisfies all the three requirements. According to experience at IBM Zurich, subgraphs with all the three characteristics above, occur frequently in business process designs [19].

One possible transformation process with the tasks T1 and T2, over four steps, is shown in Fig. 21. A model is displayed with dashed marking for elements that are replaced in the next transformation step, and $\stackrel{Tx}{\Longrightarrow}$ denotes the application of task Tx, where $x \in \{1, 2\}$.

The task T1 replaces cyclic control flow by `repeat-while` statements. The task T2 removes an activity with a single predecessor, moves its outgoing control flows to the predecessor activity, adds an `if` statement to the predecessor activity, and introduces a cyclic control flow to the predecessor activity if there is a "reverse" control flow from the successor to the predecessor. In the application of task T2 from model 1 to model 2, the `Configure` activity plays the role of a successor node with `Select` as the single predecessor activity. To ensure that there is at most one control flow in one direction between two activities, we make a combined control flow with `or` operators between the guards of the control flows.

Our chosen strategy is to allow multiple control flows in the same direction between two activities in the intermediate models, while we define separate rules to combine such multiple control flow edges into one. To simplify the example, we do not consider nested activities in this paper.

4.4.1 The transformation rules

We have defined five rules in Figs. 22, 23, 24, 25, 26 to simulate the tasks T1 and T2. Since we use the concrete syntax to define the rules, they resemble the transformation steps from Fig. 21.

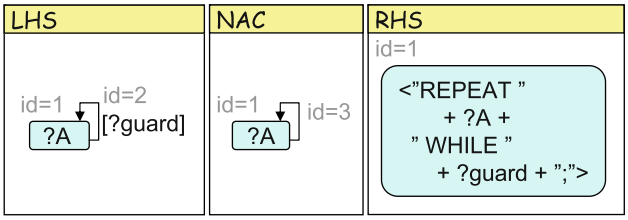


Fig. 22 *Transformation rule:* T1

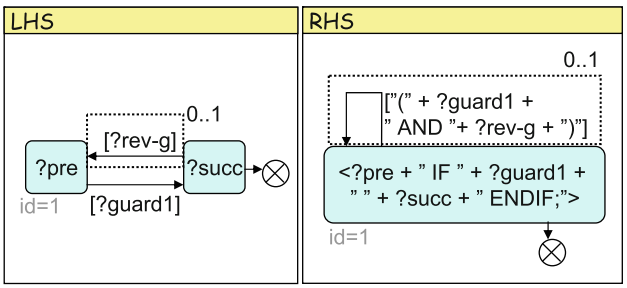


Fig. 23 *Transformation rule:* T2-NextIsFinal

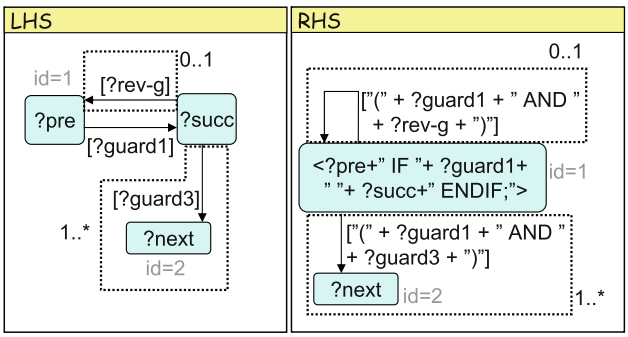


Fig. 24 *Transformation rule:* rule-T2-NextIsActivity

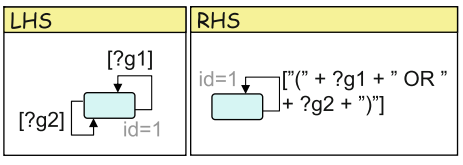


Fig. 25 *Transformation rule:* RemMultiCircEdge

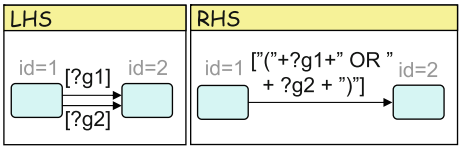


Fig. 26 *Transformation rule:* RemMultiEdge

A single rule is sufficient to simulate task T1 (Fig. 22). The LHS expresses that we are looking for matches of arbitrary activities with a cyclic control flow. `id=1` is an identifier of the matched activity, and `?guard` is a variable holding the guard value of the cyclic control flow. The NAC ensures that the matched activity has exactly one cyclic control flow. The RHS removes the cyclic control flow and extends the activity name with a `repeat-while` expression.

To simulate task T2 we define two rules depending on the node type(s) following the successor activity. Either the next node(s) is the final node or activity node(s). In both cases a "reverse" control flow going from the successor activity back to the predecessor activity shall result in a cyclic control flow of the predecessor activity, where the guards are combined with an `and` operator. A collection operator with cardinality 0..1 expresses that such a "reverse" control flow is either present or not.

For both T2 rules the predecessor activity name is extended by the same `if`-expression. For the `T2-NextIsFinal` rule (Fig. 23) the predecessor activity gets an outgoing control flow to the final node. The `T2-NextIsActivity` rule (Fig. 24) is a bit more complicated. Here, we need to move each outgoing control flow of the successor activity over to the predecessor activity, and the guard of the new control flow is extended with an `and` operator between two successive guards. A collection operator with cardinality `1..*` expresses that there are arbitrarily many such outgoing control flows from the successor activity.

Notice that we have not defined a NAC to ensure that the successor activity has exactly one predecessor activity, since this is ensured by the dangling condition. Otherwise an additional incoming control flow to the successor activity to be deleted would become a dangling edge.

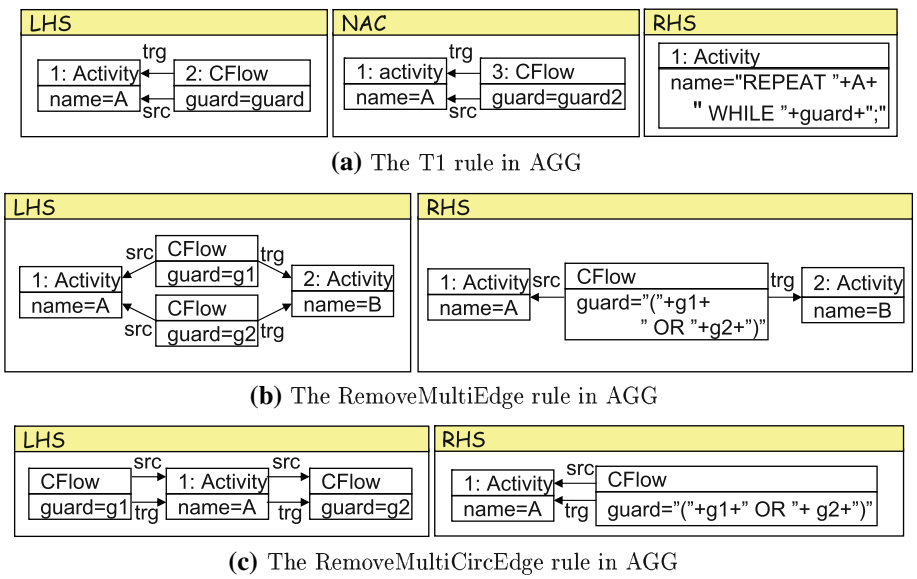
Finally, we need two simple rules to define: (1) the merging of two cyclic control flows into one control flow (`RemMultiCircEdge` shown in Fig. 25), and (2) the merging of two control flows in the same direction between two distinct activities (`RemMultiEdge` shown in Fig. 26). The merged control flow uses an `or` operator to combine the guards of the joined control flows.

The two rules to merge multiple control flow rules should always be applied after each application of a T1 or a T2 rule. However, no specific control flow ordering of the rules is necessary due to (1) the dangling condition for the two T2 rules, and (2) the NAC of the T1 rule. The NAC of the T1 rule implies that the `RemMultiCircEdge` rule must be applied as long as possible first on the relevant activity, while the dangling condition on the to-be-deleted successor activity ensures that the `RemMultiEdge` rule is applied before the T2 rules.

4.4.2 Remove unstructured cycles in AGG

We need to translate our rules from the previous section into abstract syntax so that they can be used by the AGG tool. In addition, AGG does not have a collection operator. Thus, we get several rules for a single rule with collection operator(s). An automated mapping to AGG rules is described in our earlier work [14].

In AGG, identifiers are displayed with a number followed by a colon. E.g. `1: Activity`, is shown in the rule when there are shared elements between the LHS and the RHS/NACs. Elements that are shared between the LHS and the RHS are preserved by the rule. Elements where we have not displayed an identifier occur either only in the LHS and will be deleted, or they occur only in the RHS and will be added.

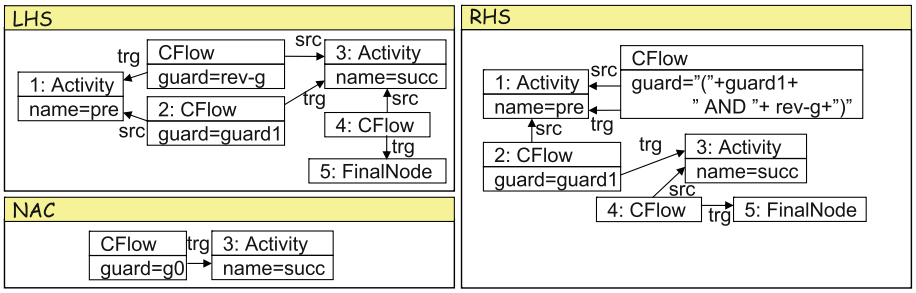


(a) The T1 rule in AGG

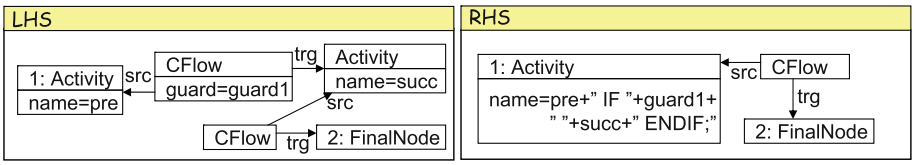
(b) The RemoveMultiEdge rule in AGG

(c) The RemoveMultiCircEdge rule in AGG

Fig. 27 Three of the AGG rules to remove unstructured cycles

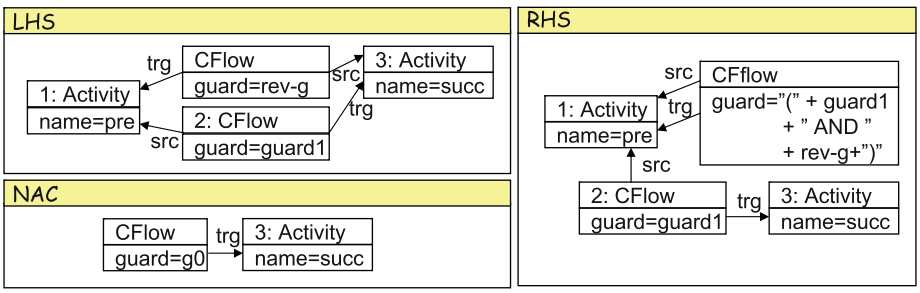


(a) The T2-NextIsFinal-Iter rule in AGG

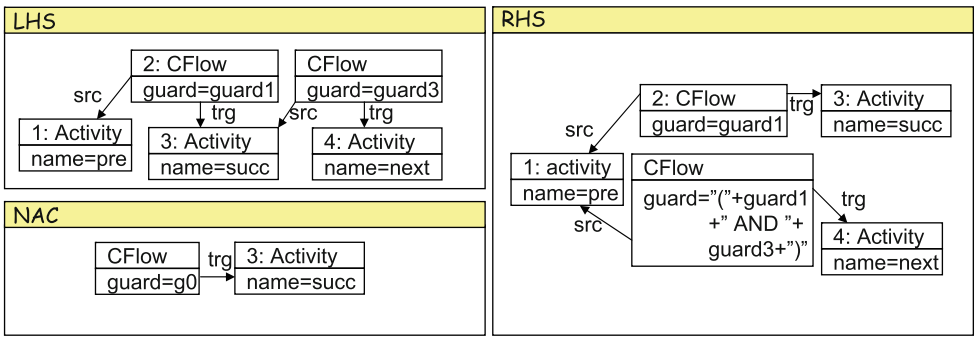


(b) The T2-NextIsFinal-Final rule in AGG

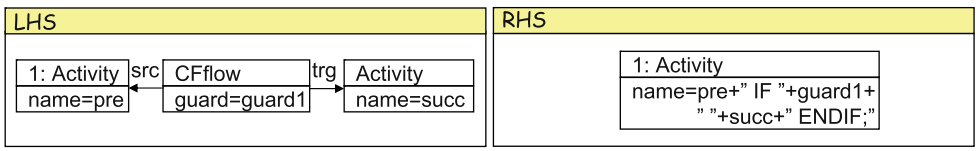
Fig. 28 Two rules are needed to simulate the T2-NextIsFinal rule in AGG



(a) The T2-NextIsActivity-Iter-1 rule in AGG



(b) The T2-NextIsActivity-Iter-2 rule in AGG



(c) The T2-NextIsActivity-Final rule in AGG

Fig. 29 Three rules are needed to simulate the T2-NextIsActivity rule in AGG

We will use the AGG rules that result directly from our tool supported mapping of concrete syntax-based and collection operator-based rules into AGG rules [14,15,32]. To be fair to AGG we manually investigated the generated rules to see if they could be optimized or further improved. No such improvements were found, and in fact the generated rules were fewer than a previous attempt where we coded the rules manually in AGG.

We get eight AGG rules (Figs. 27, 28, 29) corresponding to the five previous concrete syntax-based rules. The three previous rules without collection operators (Fig. 27) are simply translated from concrete to abstract syntax.

The rule named `T2-NextIsActivity` with two collection operators is mapped to three transactional rules in AGG. The Iter-1 rule (Fig. 29a) represents the `0..1` collection with the "reverse" control flow. The Iter-2 rule (Fig. 29b) represents the `1..*` collection with the arbitrary number of outgoing control flows from the successor activity. The Final rule (Fig. 29c) deletes the successor activity. For both the iteration rules we get autogenerated NACs to exclude matches when there are multiple predecessors for the successor activity.

The rule named `T2-NextIsFinal` with one collection operator and is mapped to two transactional rules in AGG. The Iter rule (Fig. 28a) will replace a "reverse" control flow by a cyclic control flow of the predecessor activity, and it gets a NAC to prevent multiple predecessors. The Final rule (Fig. 28b) deletes the successor activity.

In general, we need additional Java code to control the rule application order for the set of rules that come from a single rule with collection operators. The set of rules (e.g., `T2-NextIsFinal-Iter` and `T2-NextIsFinal-Final`) corresponding to a single rule with collection operators shall be applied as one transactional group The Iter rules are applied first and at least the minimum cardinality number of times, and as long as possible (or up to the maximum cardinality if it is different from $*$). The Final rule must then be applied.

In AGG, rules can be grouped in ordered layers, where rules in the first layer are applied as long as possible. Then we continue at the next layer, and this process may be repeated by looping over the layers. In many practical situations, such as the example in this section, it is sufficient to use layers and maybe add a few NACs instead of writing Java code to enforce transactional behavior of the set of collection free rules.

However, it is in many cases non-trivial to determine when a layered approach, possibly combined with NACs, is sufficient. Fujaba [12] eases this work by providing direct support for transactional rules.

The collection operator-based solution is expressed with half a paper page of graphical models, while the AGG solution needs more than one page of graphical models. In addition, AGG needs rule application order code to ensure the transactional behavior of the iteration/final rules.

5 Simulating a collection rule by collection free rules

The previous section showed how a few example collection-based rules can be simulated by collection free rules in AGG. This section presents an algorithm to map any collection-based rule to a set of collection free rules. We will also explain why this can be quite complicated. Since the collection operator is not available in AGG, we use a transactional sequence of multiple collection free rules to simulate the intended effects of a single rule with collection operators. We only consider NAC free rules in this section.

The complicated apparatus and the set of less intuitive collection free rules show the large benefits for the transformation designer to have direct support for the collection operator. The alternative is to manually define and ensure a correct execution strategy of collection free rules that simulate a single rule with collection(s)). This is time consuming and error prone.

A rule r with collection operators can be represented by zero or one Init rule, one or more Iter rules and zero or one Final rule. These rules are ordered and executed in a transaction:

- **Init rule**. The rule shall be applied only once as the first rule in the transaction. This rule has $LHS = L^{\psi^-}$, which ensures that there is a match of the original rule r. The RHS contains the LHS and the non-collection elements to be added. It must be the first applied rule, since the other rules may connect to the added elements from this rule.
- **Iter rules**. Each collection operator is mapped to an Iter rule. The Iter rule shall be applied to each subgraph match of a collection.
- **Final rule**. The Final rule deletes all non-collection elements.

Algorithm 5.1: ToCollFree(r : *CollRule*)

$$
\begin{array}{l}
nonCollAdd = r.R.remColls \setminus r.I.remColls \\
Init = \textbf{new } Rule;\ Init.L = r.L^{\psi^-}; \\
Init.R = Init.L \cup nonCollAdd \\
\textbf{for } i \leftarrow 1 \textbf{ to } r.numCollections \\
\quad \textbf{do} \begin{cases} Iter_i = \textbf{new } Rule \\ Iter_i.L = r.L.remColls \cup nonCollAdd \\ \qquad \cup\, r.L.collContent(i) \\ Iter_i.R = r.L.remColls \cup nonCollAdd \\ \qquad \cup\, r.R.collContent(i) \end{cases} \\
\textbf{if } (r.L.remColls \setminus r.I.remColls) <> \emptyset \\
\quad \textbf{then} \begin{cases} Final = \textbf{new } Rule; \\ Final.L = r.L.remColls \\ Final.R = r.R.remColls \setminus nonCollAdd \end{cases}
\end{array}
$$

The pseudocode in Algorithm 5.1 defines a mapping, as described above, from a rule with collections to a set of collection free rules, where:

- The *numCollections* method returns the number of collection operators
- The *remColls* method removes all collection operators including their content

- The *collContent*(i) method retrieves the content inside collection operator number i
- $Iter_i$ is the rule for collection i
- The rule $Iter_i$ only applies the changes relevant to collection i. By not changing any other parts, all the individual matches within collection i as well as the other collections get an equal chance to be matched.

Algorithm 5.1 does not enforce the collection cardinalities. In general, we need additional control flow to ensure that each iter rule is executed within the given collection cardinality. The iter rule shall be executed at least the lower boundary number of times and as many as possible up to the upper bound of the collection operator.

Recall Fig. 3c, which shows an example of a rule with collection operators. By following Algorithm 5.1 we get a set of rules, {*Init*, *Iter*, *Final*}, as shown in Fig. 30. The Final rule is produced since there are non-collection elements to be deleted. The Iter rule replaces a path of control flows going to the innermost decision and merge nodes, with a new path of control flows only going to the outermost decision and merge nodes with a combined guard. The Iter rule replaces one path each time the rule is applied. The Final rule is applied when the Iter rule is no longer applicable.

We need to ensure that all the rules in a single transaction involve the same context regarding the original rule's non-collection elements, which we achieve by introducing an additional `id` attribute (not shown in the figure) for all the elements. All the non-collection elements in the Iter and Final rules get id values corresponding to the elements matched by the Init rule.

Conceptually, we build an entire match based on the collection rule's LHS, and then applies the effect defined by the RHS. When simulating such a behavior with multiple rules in AGG, we need to be careful about possible dependencies and interactions between the Iter rules. One Iter rule may add elements leading to yet another individual matching of another Iter rule, which is incorrect behavior. To avoid this problem, we extend all the model elements by a Boolean helper attribute named `exclude`. Such an extension of the metamodel is always possible in AGG since we have full freedom to specify the metamodel. All `exclude` attributes are set to `false` at the start of the transaction, while all the collection content `exclude` attributes are set to `true` in the RHS of the Iter rules. Furthermore, each LHS of an Iter rule is extended so that it only matches elements with the `exclude` attribute set to `false`. By doing so, the Iter rules can be applied in an arbitrary order.

In other tools with more control flow and transaction support like in Fujaba [12] and PROGRES [33], it may be simpler than with AGG to simulate a collection rule by collection free rules. This also holds for graph transformation with recursion [16,38]. Still, the introduction of a collection operator will greatly reduce the effort needed by the transformation designer when designing rules.

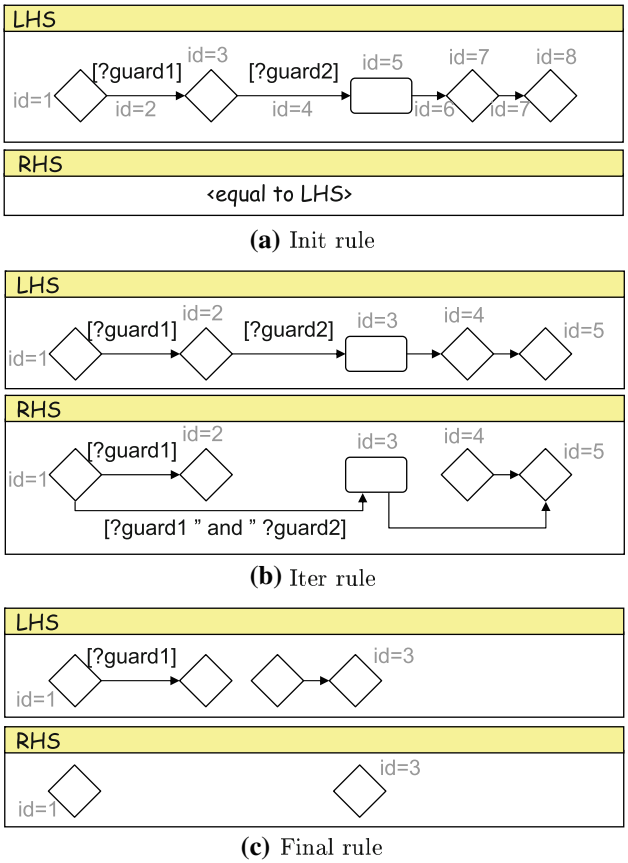


Fig. 30 Activity model refactoring (collection free)

We have developed a proof-of-concept Eclipse GMF-based [7] rule editor when activity models are used as the source and target language [14,15,32], which supports the usage of multiple collection operators in the same rule. The transformation from concrete-syntax-based rules to AGGs abstract syntax-based rules has been implemented using the MOFScript language [27]. Both the redundant decision node example (Fig. 3c) and the remove unstructured cycles example (Figs. 22, 23, 24, 25, 26) have been successfully applied in our tool. We have not implemented the transactional support needed to generally ensure a correct simulation of the collection operator.

6 Nested collection operators

In this section we extend the collection operator in order to allow nested collection operators. An example of colored Petri nets illustrates that nested collection operators can be very powerful. Colored Petri nets are extensions of basic Petri nets, where the tokens can be of a certain data type, where

the data type is referred to as the color. In the following, we use natural numbers to represent colors.

There are several variations of colored Petri nets. We use a kind of Petri nets where tokens are assigned to arcs and not only to places. An arc that goes from a preset place to a transition is referred to as a *preset arc*, and an arc that goes from a transition to a postset place is referred to as a *postset arc*.

Figure 31 shows the extensions and modifications of the concrete syntax compared to our previously presented non-colored Petri nets. We display the color of a token by an integer value next to the token. Since tokens can be assigned to arcs, we use a rectangle symbol with two edges (`src` and `trg`) so that tokens can be easily attached to the arc.

The tokens assigned to an arc will not change when transitions are fired. The tokens assigned to a preset arc indicate the type of tokens that are removed from the corresponding preset place when firing a transition. Correspondingly, the assigned tokens to a postset arc indicate the type of tokens that are assigned to the corresponding postset place, when firing a transition. A transition is enabled to fire if and only if all of the transition's preset places are able to deliver the tokens that are required by their corresponding preset arcs, i.e., a preset place must contain at least the same tokens as its preset arc.

Figure 32a shows a source model of a colored Petri net, and Fig. 32b shows the resulting model after firing a transition. We explicitly show an identifier of an arc by a number prefix so that the explanatory text can easily refer to different parts of the model.

Figure 33 defines a single transformation rule to express a transition firing using nested collection operators. Each of the two outermost collection operators contains an inner collection operator. Collection `c1` matches all the preset arcs and preset places of a transition. The inner collection `c2` matches all those tokens that are shared by a preset arc and its corresponding preset place. The NAC also contains the `c1` collection, which means that there is one NAC for each preset arc. There is an assigned token to each of these arcs. This token does not have a shared identifier with the LHS tokens, and thus the NAC ensures that there cannot be any unmatched tokens assigned to any of the preset arcs. Hence, the LHS and the NAC together capture the condition for when a transition firing is enabled.

In the RHS of the rule, collection `c2` has been reduced. The effect is that the rule removes tokens similar to all the preset arc assigned tokens, from the corresponding preset places.

Collection `c3` matches all the postset arcs and places of a transition. The inner collection `c4` matches all the tokens that are assigned to each postset arc. In the RHS of the rule, collection `c4` has been increased. The effect is that each postset place gets new tokens similar to the tokens of its corresponding arc.

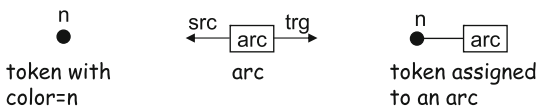


Fig. 31 Concrete syntax for colored Petri nets

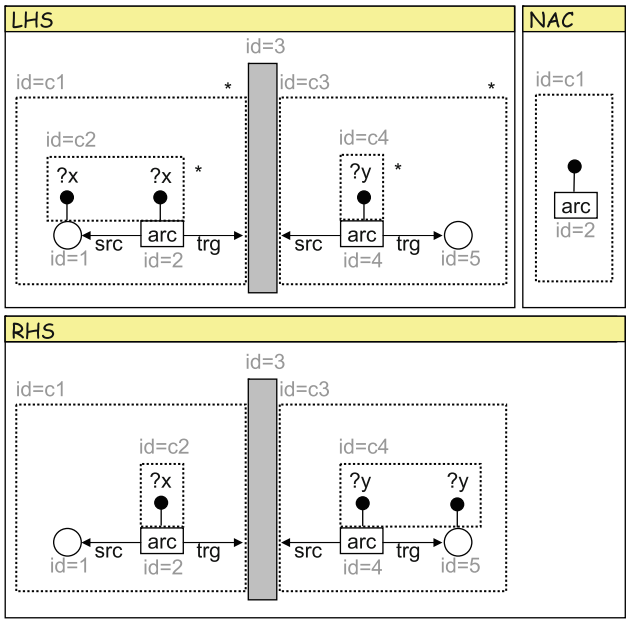
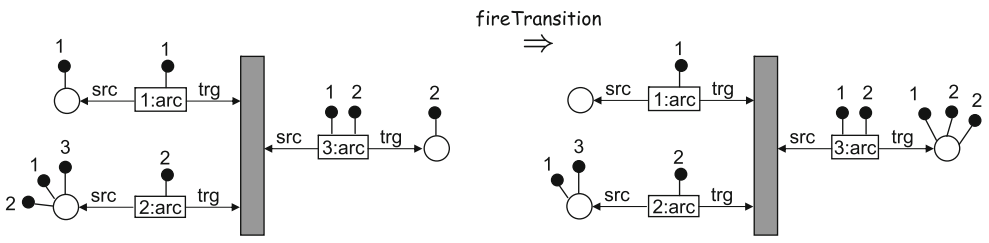


Fig. 33 Rule to fire transition in a colored Petri net



(a) Source model **(b)** Target model produced by firing a transition

Fig. 32 Fire transition in a colored Petri net

A question mark prefix indicates a variable which matches any value. The variable `?x` is used twice as token color values within collection `c2`, which means that they must be matched to the same value for each match of the collection. The variable `?y` is used as a token color value in the LHS within collection `c4`. The RHS adds a token with the same color `?y` to the corresponding postset place.

6.1 Nested collection operators formalized

In this section we will extend the previous formalization of the collection operator (Sect. 3.3) to include support for nested collection operators. Let $c.lev$ denote the nesting level of a collection operator c, and let $par(c)$ denote the immediately enclosing parent collection operator if $c.lev \geq 2$. The following requirement expresses that the nesting structure must be preserved within the three rule graphs L,I and R.

Requirement 6 (Preserved nesting structure) *Given a rule $L \xleftarrow{l} I \xrightarrow{r} R$. Each interface collection operator must be at the same level in the three rule graphs L,I and R, and their nesting structure must be preserved. Formally,*

$$\begin{aligned}\forall c \in Coll_I \; : \; & l(c).lev = c.lev = r(c).lev \\ & \wedge \; (c.lev = 1 \vee (par(l(c)) = l(par(c)) \\ & \wedge \; par(r(c)) = r(par(c)))\end{aligned}$$

As explained in Sect. 3.3, the LHS introduces the set of all collection operators in a rule, and a ψ mapping is completely defined when it is defined for all the LHS collection operators. This holds due to Requirements 1, 2 and 6. Thus, we will again concentrate on the LHS (L) for a rule $L \leftarrow I \rightarrow R$ in the following definitions. First, we define the following notations and helper functions:

- $L.maxLev$ is the maximum number of nesting levels of nested collection operators. For the colored Petri net rule (Fig. 33): $L.maxLev = 2$
- $Coll_L[1]$ is the set of all the outermost collection operators, i.e., the collection operators at nesting level 1. For the colored Petri net rule: $Coll_L\,[1] = \{c1, c3\}$.
- $\psi[1]$ is the cardinality mappings for all the collection operators at nesting level 1, i.e., $\psi[1] \; : \; Coll_L[1] \rightarrow (\mathbf{N} = \{0, 1, 2, \ldots\})$, where $\forall c \in Coll_L[1] : \psi[1](c) \in [c.min, c.max]$. For the colored Petri net rule, one example is $\psi[1] = \{c1 \rightarrow 2, c3 \rightarrow 1\}$.
- $Coll_L^{\psi[i-1]}[i]$, where $i \in \{2, L.maxLev\}$ is the set of all the collection operators at nesting level i. The number of collection operators at nesting level i depends on the cardinality mappings at the previous nesting level $i-1$, which is why $\psi[i-1]$ is included as superscript in the notation. For the colored Petri net rule and the $\psi[1]$ assignment from the previous item:

  $$Coll_L^{\psi[1]}[2] = \{c2(1), c2(2), c4(1)\}$$

  where $c2(k)$ denotes collection operator $c2$ for the kth match of the enclosing collection operator $c1$, and $c4(k)$ denotes collection operator $c4$ for the kth match of the enclosing collection operator $c3$. There are two $c2$ collection operators since $c2$ is nested within $c1$ and $(c1 \rightarrow 2) \in \psi[1]$. Similarly, there is one $c4$ collection operator.
- $Coll_L^{\psi[i]}$ is the set of all collection operators at the nesting levels from 1 to i (any additional collection operators at nesting levels $[i+1, L.maxLev]$ are not part of this set), i.e.,

  $$Coll_L^{\psi[i]} = Coll_L\,[1] \cup \bigcup_{j=2}^{i} Coll_L^{\psi[j-1]}[j]$$

- $Coll_L^{\psi}$ is the set of all the collection operators *at all nesting levels*, i.e.,

  $$Coll_L^{\psi} = Coll_L\,[1] \; \cup \bigcup_{i=2}^{L.maxLev} Coll_L^{\psi[i-1]}[i]$$

- $\psi[i]$ is the cardinality mappings for all the collection operators at nesting level i, i.e.,

  $$\psi\,[i] : Coll_L^{\psi[i-1]}[i] \rightarrow (\mathbf{N} = \{0, 1, 2, \ldots\})$$

  where $i \in [2, L.maxLev] \; \wedge \; \forall c \in Coll_L^{\psi[i-1]}[i] : \psi[i](c) \in [c.min, c.max]$
- ψ is the cardinality mappings for all the collection operators at all nesting levels, i.e.,

  $$\psi = \bigcup_{i=1}^{L.maxLev} \psi[i]$$

Definition 6, which defines cardinality mapper extensions, needs to be updated to be applicable also when we have nested collection operators. Definitions 11–13 define the necessary changes. Definition 11 defines equality of two cardinality mappers. Definition 12 uses Definition 11 to define extension of a cardinality mapper for a fixed nesting level. Definition 13 uses Definition 12 in the general definition of cardinality mapper extension.

Definition 11 *(Fixed level mapping equality)* Two cardinality mappers ψ_1 and ψ_2 for a rule $L \leftarrow I \rightarrow R$ are equal at a fixed nesting level i (denoted $\psi_1 =_L^i \psi_2$) if and only if their sets of collection operators are the same up to nesting level i

and they are all mapped to the same numbers. Formally,

$$\psi_1 =^{i}_{L} \psi_2 \stackrel{\text{def}}{=} Coll^{\psi_1[i]}_{L} = Coll^{\psi_2[i]}_{L} \wedge \forall c \in Coll^{\psi_1[i]}_{L} : \psi_1(c) = \psi_2(c)$$

Definition 12 (Fixed level mapping extension) A cardinality mapper ψ^+ extends ψ for a rule $L \leftarrow I \rightarrow R$ at a fixed nesting level i (denoted $\psi^+ \succ^{i}_{L} \psi$) if and only if there is a nesting level where all the following three conditions are satisfied: (1) there is a collection operator with larger mapping, and (2) all the other collection operators on that level are at least as large, and (3) all collection operators in lower nesting levels are equally large. Formally,

$$\psi^+ \succ^{i}_{L} \psi \stackrel{\text{def}}{=}$$
if $i = 1$ **then**
$\exists c \in Coll_L[1] : \psi^+(c) > \psi(c) \wedge$
$\forall c \in Coll_L[1] : \psi^+(c) \geq \psi(c)$
else $(\psi^+ \succ^{i-1}_{L} \psi) \vee$
$(\psi^+ =^{i-1}_{L} \psi \wedge$
$\exists c \in Coll^{\psi[i-1]}_{L}[i] : \psi^+(c) > \psi(c) \wedge$
$\forall c \in Coll^{\psi[i-1]}_{L}[i] : \psi^+(c) \geq \psi(c))$

Definition 13 *(Extension for nested collection operators)* Given a rule $L \leftarrow I \rightarrow R$. A cardinality mapper ψ^+ extends the cardinality mapper ψ (denoted $\psi^+ \succ_L \psi$) if and only if ψ^+ extends ψ at the maximum nesting level. Formally,

$$\psi^+ \succ_L \psi \stackrel{\text{def}}{=}$$

$$\psi^+ \succ^{L.maxLev}_{L} \psi$$

The remaining parts of Definitions 6 and 10 of extensions and matching need no changes, and we have formally defined when a match for a rule with nested collection operators is applicable. Requirement 4 that expresses 'non-overlapping matches' must be adjusted so that it applies only when the collection operators `c1` and `c2` are at the same nesting level.

6.2 The matching process for nested collection operators

We now describe the matching and transformation process for rules with nested collection operators. $\psi^-[i]$ denotes the minimal configuration of ψ of the remaining unassigned collection cardinalities, assuming that all cardinalities at lower nesting levels ($j < i$) have been assigned, i.e.,

$$\forall c \in \bigcup_{j=i}^{L.maxLev} Coll^{\psi[j-1]}_{L}[j] : \psi(c) = c.min$$

Given a rule $p : L \leftarrow I \rightarrow R$ with collection operators and a graph G, the following steps can be used to find a cMatch in p and try to apply a derivation step for that cMatch:

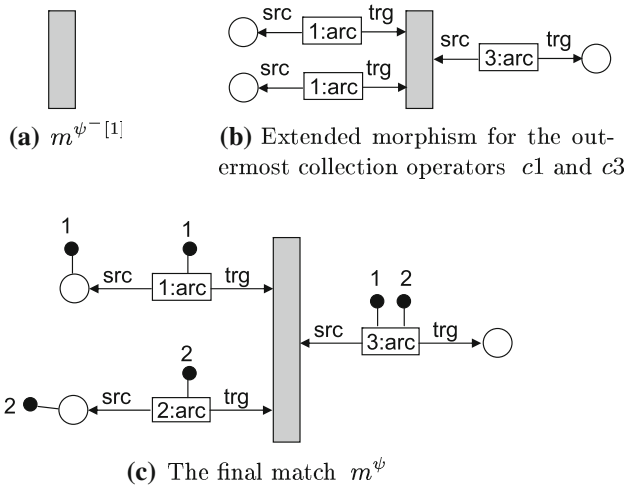
1. Look for an injective morphism $m^{\psi^-[1]} : L^{\psi^-[1]} \rightarrow G$ in the collection free rule $p^{\psi^-[1]}$.
2. Let i iterate from nesting level 1 to $L.maxLev$. Extend the matches at nesting level i as much as possible, while ensuring that these extensions have $\psi^-[i+1]$ cardinalities for all the nested collection operators. The extension process can be achieved by iterating over each collection operator c at nesting level i and increasing $\psi(c)$ as much as possible. $\psi(c)$ can only be increased by 1, if the injective morphism can be extended with an additional subgraph match of the collection content in c.
3. Finally, we have reached a non-extendable match m^{ψ}. Apply a derivation step with the collection free rule p^{ψ} if the match m^{ψ} satisfies all the NACs and the dangling condition.

Figure 34 explains the matching process, over three steps, for our colored Petri net example and the source model from Fig. 32a.

Step 1. First, we look for an injective morphism where all collection operator cardinalities are equal to their minimum cardinality, which for our rule is zero for all collection operators. The resulting morphism, $m^{\psi^-[1]}$, consists of the single transition node in the model (Fig. 34a).

Step 2. We continue by extending all the collection operators at nesting level 1, and the updated morphism is the graph shown in Fig. 34b. The temporary cardinality mapping is $\psi = \{c1 \rightarrow 2, c3 \rightarrow 1, c2(1) \rightarrow 0, c2(2) \rightarrow 0, c4(1) \rightarrow 0\}$.

Step 3. We proceed by extending the five collection operators as much as possible at nesting level 2, resulting in the final cardinality mapping $\psi = \{c1 \rightarrow 2, c3 \rightarrow 1, c2(1) \rightarrow 1, c2(2) \rightarrow 1, c4(1) \rightarrow 2\}$. The final match result (Fig. 34c) is then a non-extendable match $m^{\psi} : L^{\psi} \rightarrow G$. The matching process has dynamically built a collection free rule p^{ψ} (shown in Fig. 35) for the ψ mappings.



(a) $m^{\psi^-[1]}$ **(b)** Extended morphism for the outermost collection operators $c1$ and $c3$

(c) The final match m^{ψ}

Fig. 34 The matching process for the colored Petri net rule

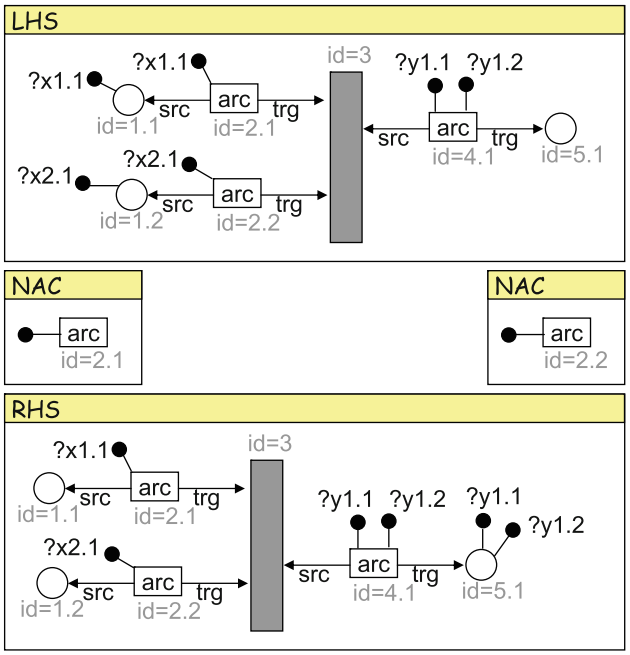
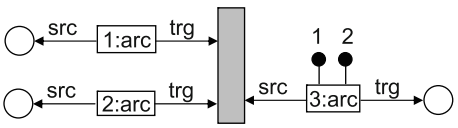
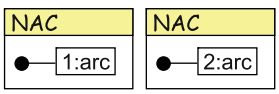


Fig. 35 Dynamically built collection free rule for firing a transition in a colored Petri net



(a) cMatch in the model from Figure 32b



(b) NACs

Fig. 36 When the transition is disabled to fire, the dynamically built NACs ensure that the derivation step cannot be applied

The matching process has produced two NACs, one for each preset arc. We can apply a derivation step since (1) all the three NACs are satisfied, and (2) the dangling condition is satisfied, i.e., application of the rule p^{ψ} on the match m^{ψ} does not lead to any dangling edges. The result of the derivation step is the model in Fig. 32b. Upon this model we now continue the matching process.

Figure 36a shows the obtained non-extendable match, i.e., a cMatch, for this model. This time, none of the two NACs (Fig. 36b) are satisfied, since both preset arcs have unmatched assigned tokens. This means that we cannot apply the derivation step, and this is the expected result that corresponds to the transition being disabled for firing.

This section and Sect. 3.4 describe the matching process in detail, in which we dynamically build a single collection free rule. We recommend to implement tool support for the collection operator by following this matching process instead of the translation into a set of transactional collection free rules which are fixed at compile time. The latter strategy is complicated as explained in Sect. 5.

7 Related work

This paper has extended our earlier work [14] where the collection operator was restricted to the transformation of activity models. The improvements in this paper include support for multiple collection operators of arbitrary lower and upper bound cardinalities and nested collection operators, in the same rule.

A conference paper with the same title [13] has been revised and considerably extended by this paper. The most important extension is nested collection operators. In addition, the large example given in Sect. 4.4 is taken from another conference paper [15].

In this section we describe related approaches, and these can be categorized into three groups: Sect. 7.1, added expressive power to traditional graph transformation; Sect. 7.2, collection matching and transformation that is restricted to single nodes only; and Sect. 7.3, collection matching and transformation of subgraphs.

7.1 Graph transformation extensions

Guerra and de Lara [16] have added recursion to the algebraic graph transformation formalism with an associated user-friendly notation. A recursion rule consists of two match criteria graphs, a base and a recursion graph, and an effect graph to express the changes. The effect graph is visualized as a single graph that combines the LHS, Interface and RHS graphs from algebraic graph transformation.

Their approach is complementary to our collection operator. The recursion construct can be used to express matching and transformation of a graph structure with a recursive nature such as a repetitive path, e.g., an arbitrary long inheritance structure in UML class models. Such matching cannot be expressed by our collection operator.

On the other hand, the recursion construct cannot, in general, replace the collection operator. A single recursion-based rule cannot handle any of the examples in this paper. The recursion operator works iteratively by applying a similar rule on each individual match, while the collection operator collects all individual matches in one large group before the rule is applied. This means that the recursion construct is not suited when the rule deletes or adds content in the non-collection part. This problem occurs for the example in Fig. 3c which deletes several non-collection elements and for the example in Fig. 19 which adds a non-collection element.

Furthermore, a NAC condition must apply to all individual matches and cannot be seen in relation to the entire matches of a collection. This problem occurs for the examples in Figs. 13 and 18. In addition, one recursion rule is needed for each collection operator in a single rule with multiple collection operators.

Recursion has also been added to the graph transformation tool VIATRA2 [3,38]. Bergmann et al. [3] show how nested NACs can be used to simulate a NAC in relation to collection content. This is used in [3] to express the example of firing transitions in a Petri net (Fig. 18). However, they still need three rules to replace our single graphical rule. Furthermore, they need additional control flow to iterate over elements and to call the three rules. One rule expresses the firing condition, elegantly by nested NACs, and the last two rules express removal and addition of tokens, respectively.

Lawley and Steel [21] have support for recursion in Tefkat, which is a textual language for specifying model transformations. As with the other recursion-based alternatives, recursion does not directly address the needs addressed by a collection operator. Hence, multiple rules and additional control flow is needed in general to simulate single collection operator-based rules. Tefkat is not intended for model updates or refactorings such as we have in all the examples except in Fig. 20. It is an interesting future path of research to investigate if textual languages like Tefkat can be improved by a new textual construct in a similar way as the collection operator for graphical languages.

Lindqvist et al. [23] propose the star operator, which, like recursion, is suited to find repetitive occurrences of a specific modeling pattern. The star operator is only defined for matching model extracts, and not to do transformations.

7.2 Single node collections

Fujaba [12] and PROGRES [33] have support for matching collections of single nodes only (*set nodes* in PROGRES, *multi objects* in Fujaba), which is a limited expressiveness compared to the collection operator that allows for collections of a fixed but arbitrarily large subgraph. Furthermore, the single node approaches are only defined for abstract syntax. To determine if single node collections are expressive enough for a particular transformation task may depend on the choice of abstract syntax representation of the involved source and target languages.

As an example, we now consider if we can use single node collections to express a rule for firing of Petri nets (Fig. 18). If the abstract syntax of Petri net graph representation uses two different node types to represent tokens and places, then a rule to perform transition firing with single node collections will fail. This is because all tokens of the places in the transition preset will be consumed, and not only one token per place as required. This problem can be avoided by choosing a different abstract syntax where a place has an integer attribute to keep track of the number of tokens instead of having a separate node type for a token. In general, it is undesirable to adjust the abstract syntax due to limitations in the rule language. By using *E-graphs* [10] where edges can have attributes we can get away with using single node collections for some, but not all, of the paper examples, depending on the choice of abstract syntax.

7.3 Subgraph collections

The approaches in this section are all capable of handling subgraph collection matching and transformation.

Amalgamated rules by Taentzer et al. [6,36] can simulate the collection operator. Our collection operator is more concise since we can use a single rule, while they need one *subrule* to capture the rule part outside of all collections, and one *elementary rule* for each collection operator.

The remaining approaches discussed in this section have all been worked out in parallel with our work.

A *group operator*, introduced by Balasubramanian et al. [1] and implemented in the GREaT tool, enables arbitrarily large subgraph matches that can be copied, moved or deleted. However, the subgraph matches can not be modified as with our collection operator.

Nested quantification is proposed by Rensink [30] as an extension to the GROOVE tool, which is similarly concise as our collection operator by allowing a single rule to express subgraph matches. His notation is a bit different from ours since they use exists (∃) and for all (∀) quantifiers to express the parts outside of a collection, and those inside a collection respectively.

Fuss and Tuttlies [11] propose an extension to PROGRES called *set-regions*, which is quite similar to our collection operator. However, the concrete notation of such set-regions within the rules is not shown.

The set regions from Fuss and Tuttlies [11] and nested quantification from Rensink and Kuperus [31] can be nested as with our collection operator. Fuss and Tuttlies, however, do not provide details on how this can be implemented. Rensink and Kuperus [6] have a relatively complicated formalization of their nesting support, which is based on the rule amalgamation technique. Our nesting support, on the other hand, dynamically builds a collection free rule in the matching process and then reuses the existing apparatus of algebraic graph transformation. Rensink and Kuperus use an example of repotting flowering geraniums to illustrate the benefit of nesting. The repotting of geraniums can easily be expressed by using one outermost collection operator with a nested collection operator. The outermost collection operator enclose all the elements of the LHS and RHS of the rule. In such cases we can quite easily manage also without nesting, by removing the outermost collection operator and apply the rule in a

separate layer as-long-as-possible. Our example of colored Petri nets is a better justification why it is useful to include nesting support.

Minas and Hoffmann [18,24] define a *cloning operator* which is an alternative to our collection operator. Cloned nodes and incident edges correspond to elements inside a collection operator. They support multiple elements inside the same collection operator by assigning the same cloning identifier to several cloned nodes (the incident edges of the cloned nodes implicitly belongs to the same collection).

To our best knowledge none of the other subgraph collection matching approaches have support for shared variables nor collection cardinalities beyond 0..∗ and 1..∗. Furthermore, the other approaches focus only on applying their collection operators on the abstract syntax. The notations by Rensink [30] and as sketched by Fuss and Tuttlies [11], however, have a nature which makes them appropriate to be introduced on the concrete syntax, which is not the case for Minas and Hoffmann [18,24].

7.4 Related work summary

As a summary of related work we provide a table in Fig. 37. Each column in the table contains a collection operator-based transformation rule example from this paper.

Each row in the table contains three representative constructs from each related work section above. For each construct we check if it can directly express the transformation rule examples in this paper using a single rule, as we can do with our collection operator.

For several reasons mentioned above none of the paper examples can be expressed by using the recursion construct [16]. The examples in Figs. 13 and 18 cannot be expressed since they involve a NAC related to collection content. The example in Fig. 18 has the additional problem that we need to keep track of the already processed preset and postset places. The example in Fig. 33 cannot be expressed for two reasons. Recursion does not support nested collection operators and it does not support a NAC related to collection content.

Although none of the examples can be expressed by a single recursion rule, some of them can be expressed using multiple rules. The number of needed rules are indicated in parentheses in the respective table cells. The example in Fig. 3c can be expressed by one recursion rule and one rule to delete non-collection content after the recursion rule. The example in Fig. 19 can be expressed by one rule to do addition of non-collection content before a second recursion rule is applied. The example in Fig. 20 can be expressed by one rule to do addition of non-collection content before two recursion rules are applied in any order. The example in Fig. 23 can be expressed by two rules and the example in Fig. 24 can be expressed by three rules.

For the Fujaba multi object construct [12], there are several examples that are marked as (√). This indicates that they can be supported only for certain graph representations. For some of the examples a control flow edge needs to be represented by a graph edge and the guard must be an attribute of the edge. The set node cannot be used if the control flow edge is represented by a graph node.

The nested quantification [30] supports all the examples except those in Figs. 23 and 24. This is due to the unsupported collection cardinality 0..1. Instead we need two quite similar rules for each of the two examples to handle the 0 cardinality cases and the 1 cardinality cases, respectively. The example in Fig. 19 also uses a cardinality which is unsupported by other constructs (2..∗). However, this rule can easily be simulated by having two explicit non-collection elements combined with the collection as before, except for changing the cardinality to 0..∗.

When there is no direct support for the collection operator, it needs to be simulated by multiple collection free rules in a transactional manner. Programmable graph transformation approaches such as Fujaba and PROGRES can in many cases simulate the required behavior. In principle, such a simulation will correspond to the collection free simulation we have shown in Sect. 5. For the transformation rule designer, however, this is far more complicated than having support for a collection operator in the rules.

construct \ example	Fig.3c	Fig.13	Fig.18	Fig.19	Fig.20	Fig.23 and Fig.24	Fig.33
Recursion [15]	– (2)	–	–	– (2)	– (3)	– (2 + 3)	–
Fujaba multi-object [11]	(✓)	(✓)	(✓)	(✓)	–	(✓)	–
Nested Quantification [28]	✓	✓	✓	✓	✓	–	✓

Fig. 37 Table showing to which extent three other constructs can express transformation tasks similar to the examples (indicated by figure references) where a single rule with collection operator(s) was sufficient

8 Conclusions

In this paper we have introduced the collection operator, which makes graph transformation suitable to use on a number of model transformation cases where it would be cumbersome or impractical without. The collection operator raises the level of abstraction, which is a benefit for the transformation designer. For model transformations where the collection operator naturally applies, Sect. 5 shows that it is a complicated and time consuming task to manually define transformations without the collection operator.

The collection operator can be used both on the concrete syntax of the modeling language and at the abstract syntax of graphs. A single transformation rule can have multiple collection operators and they can be nested. A straightforward matching and transformation strategy is described in Sect. 3.4, and it is extended in Sect. 6.2 for nested collection operators. Our matching process dynamically builds a collection free graph transformation rule and then reuses the existing algebraic graph transformation apparatus.

We leave it as future work to provide full support for the collection operator within a graph transformation tool, and to investigate how the use of collection operators affect the theory of termination and confluence.

Acknowledgments The work reported in this paper has been funded by The Research Council of Norway, grant no. 167172/V30 (the SWAT project), and by the DiVA project Grant no. 215412 (EU FP7 STREP).

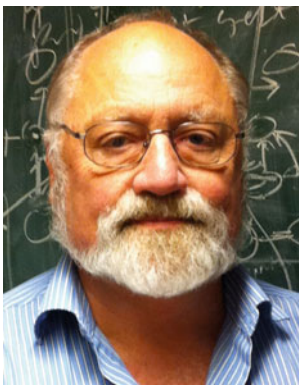
References

1. Balasubramanian, D., Narayanan, A., Neema, S., Shi, F., Thibodeaux, R., Karsai, G.: A Subgraph Operator for Graph Transformation Languages. ECEASST, 6 (2007)
2. Batory, D.S.: Feature models, grammars, and propositional formulas. In: Software Product Lines, 9th International Conference, SPLC, vol. 3714. Lecture Notes in Computer Science. Springer, Berlin (2005)
3. Bergmann, G., Horváth, Á., Ráth, I., Varró, D.: A Benchmark evaluation of incremental pattern matching in graph transformation. In: Graph Transformations, 4th International Conference, ICGT, vol. 5214. Lecture Notes in Computer Science, pp. 396–410. Springer, Berlin (2008)
4. Biermann, E., Ermel, C., Hurrelmann, J., Ehrig, K.: Flexible visualization of automatic simulation based on structured graph transformation. In: IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC (2008)
5. Biermann, E., Ermel, C., Taentzer, G.: Precise semantics of EMF model transformations by graph transformation. In: Model Driven Engineering Languages and Systems, 11th International Conference, MoDELS (2008)
6. Jaramillo J.de Lara , Ermel C., Taentzer G., Ehrig K. (2004) Parallel graph transformation for model simulation applied to timed transition petri nets. Electr. Notes Theor. Comput. Sci. 109:17–29
7. Eclipse Consortium.: Eclipse Graphical Modeling Framework (GMF) (2007). http://www.eclipse.org/gmf
8. Eder, J., Gruber, W., Pichler, H.: Transforming workflow graphs. In: Conference on Interoperability of Enterprise Software and Applications (2005)
9. Ehrig, H., Ehrig, K., Ermel, C., Hermann, F., Taentzer, G.: Information preserving bidirectional model transformations. In: Fundamental Approaches to Software Engineering (FASE). Lecture Notes in Computer Science. Springer, Berlin (2007)
10. Ehrig, H., Prange, U., Taentzer, G.: Fundamental theory for typed attributed graph transformation. In: Graph Transformations, Second International Conference, ICGT. Lecture Notes in Computer Science. Springer, Berlin (2004)
11. Fuss, C., Tuttlies, V.E.: Simulating set-valued transformations with algorithmic graph transformation languages. In: Applications of Graph Transformations with Industrial Relevance (AGTIVE). Lecture Notes in Computer Science. Springer, Berlin (2008)
12. Geiger, L., Zündorf, A.: Tool Modeling with Fujaba. Electr. Notes Theor. Comput. Sci. **148**(1) (2006)
13. Grønmo, R., Krogdahl, S., Møller-Pedersen, B.: A collection operator for graph transformation. In: Theory and Practice of Model Transformations, Second International Conference, ICMT, vol. 5563. Lecture Notes in Computer Science. Springer, Berlin (2009)
14. Grønmo R., Møller-Pedersen B.: Aspect diagrams for UML activity models. In: Applications of Graph Transformations with Industrial Relevance, Third International Symposium, AGTIVE 2007, Revised Selected and Invited Papers, vol. 5088. Lecture Notes in Computer Science. Springer, Berlin (2008)
15. Grønmo, R., Møller-Pedersen, B., Olsen, G.K.: Comparison of three model transformation languages. In: Model Driven Architecture—Foundations and Applications, 5th European Conference, ECMDA-FA, vol. 5562. Lecture Notes in Computer Science. Springer, Berlin (2009)
16. Guerra, E.,de Lara, J.: Adding Recursion to Graph Transformation. ECEASST, 6 (2007)
17. Heckel, R., Küster, J.M., Taentzer, G.: Confluence of typed attributed graph transformation systems. In: Graph Transformation, First International Conference, ICGT (2002)
18. Hoffmann B., Janssens D., Eetvelde N.V. (2006) Cloning and Expanding Graph Transformation Rules for Refactoring. Electr. Notes Theor. Comput. Sci. 152:53–67
19. Koehler, J., Hauser R., Sendall, S., Wahler, M.: Declarative techniques for model-driven business process integration. IBM Syst. J. **44**(1) (2005)
20. Lambers, L., Ehrig, H., Orejas, F.: Conflict detection for graph transformation with negative application conditions. In: Graph Transformations, Third International Conference, ICGT. Lecture Notes in Computer Science. Springer, Berlin (2006)
21. Lawley, M., Steel, J.: Practical declarative model transformation with Tefkat. In: MoDELS Satellite Events, vol. 3844. Lecture Notes in Computer Science, pp. 139–150. Springer, Berlin (2006)
22. Levendovszky, T., Prange, U., Ehrig, H.: Termination criteria for DPO transformations with injective matches. Electr. Notes Theor. Comput. Sci. **175**(4) (2007)
23. Lindqvist, J., Lundkvist, T., Porres, I.: A query language with the star operator. In: Workshop on Graph Transformation and Visual Modeling Techniques (2007)
24. Minas M., Hoffmann B. (2008) An Example of Cloning Graph Transformation Rules for Programming. Electr. Notes Theor. Comput. Sci. 211:241–250
25. Object Management Group: UML 2.0 Superstructure Specification, OMG Adopted Specification ptc/03-08-02, August 2003

26. Object Management Group: Business Process Modeling Notation (BPMN) Version 1.0, May 2004
27. Oldevik, J., Neple, T., Grønmo, R., Aagedal, J.Ø., Berre, A.-J.: Toward standardised model to text transformations. In: Model Driven Architecture—Foundations and Applications, First European Conference, ECMDA-FA (2005)
28. Ouyang, C., Dumas, M., van der Aalst, W.M.P., ter Hofstede, A.H.M., Mendling, J.: From business process models to process-oriented software systems. ACM Trans. Softw. Eng. Methodol. **19**(1) (2009)
29. Plump, D.: Confluence of graph transformation revisited. In: Processes, Terms and Cycles: Steps on the Road to Infinity, Essays Dedicated to Jan Willem Klop, on the Occasion of His 60th Birthday. Lecture Notes in Computer Science, pp. 280–308. Springer, Berlin (2005)
30. Rensink, A.: Nested quantification in graph transformation rules. In: Graph Transformations, Third International Conference, ICGT. Lecture Notes in Computer Science. Springer, Berlin (2006)
31. Rensink, A., Kuperus, J.-H.: Repotting the Geraniums: on nested graph transformation rules. In: Proceedings of the 8th International Workshop on Graph Transformation and Visual Modeling Techniques (2009)
32. Grønmo, R.: Proof-of-concept Model Transformation Tool for UML Activity Models with Support for the Collection Operator. http://folk.uio.no/roygr/ECMDA-2009-impl.zip
33. Schürr, A., Winter, A.J., Zündorf, A.: Graph grammar engineering with PROGRES. In: 5th European Software Engineering Conference. Lecture Notes in Computer Science. Springer, Berlin (1995)
34. Skogan, D., Grønmo, R., Solheim, I.: Web service composition in UML. In: Proceedings of the 8th IEEE International Enterprise Distributed Object Computing Conf (EDOC'04), Monterey, CA, September 2004
35. Sunyé, G., Pollet, D., Traon, Y.L., Jézéquel, J.-M.: Refactoring UML models. In: The Unified Modeling Language, Modeling Languages, Concepts, and Tools. Lecture Notes in Computer Science. Springer, Berlin (2001)
36. Taentzer, G.: Parallel and distributed graph transformation. Formal description and application to communication-based systems. PhD thesis, Technische Universität Berlin (1996)
37. Taentzer, G.: AGG: A graph transformation environment for modeling and validation of software. In: Applications of Graph Transformations with Industrial Relevance, Second International Workshop (AGTIVE), 2003
38. Varró, G., Horváth, Á., Varró, D.: Recursive graph pattern matching. In: Applications of Graph Transformations with Industrial Relevance, Third International Symposium, AGTIVE, vol. 5088. Lecture Notes in Computer Science. Springer, Berlin (2008)
39. Whittle, J., Jayaraman, P., Elkhodary, A., Moreira, A., Araújo, J.: MATA: A unified approach for composing UML aspect models based on graph transformation. Trans. Aspect Oriented Softw. Dev. VI **5560** (special issue on Aspects and Model-Driven Engineering, 2009)

Author Biographies

Roy Grønmo has been working as a research scientist at SINTEF since 1998. He holds a doctor degree in Computer science at the University of Oslo. He has numerous publications and programme committee memberships in international conferences and journals. The main research topics are model-driven development, model and graph transformation, service-oriented modeling and aspect-oriented modeling.



Stein Krogdahl is a professor at the University of Oslo, and got his Cand. Real. degree at the same university in 1973. His research interests include programming languages and their implementation, combinatorial algorithms, and program verification. He is currently involved in work on Package Templates, which can be seen as a static version of virtual classes.

Birger Møller-Pedersen is a professor at University of Oslo. He has worked with object orientation, from various implementations of SIMULA to the design of BETA. He was a key person in adding object-orientation to ITU SDL (standardized 1992). With Ericsson he contributed to UML 2.0 within OMG.